December 14, 2001

To the Graduate School:

This thesis entitled “Design and Analysis of a Network Transfer Layer for Parallel File Systems” and written by Philip H. Carns is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Engineering.

Walter B. Ligon III, Advisor

We have reviewed this thesis
and recommend its acceptance:

Ron Sass

Adam Hoover

Accepted for the Graduate School:

Design and Analysis of a Network Transfer Layer for Parallel File Systems

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Engineering

by

Philip H. Carns

December 2001

Advisor: Dr. Walter B. Ligon III

Abstract

This document describes the design and analysis of a network transfer layer for the Parallel Virtual File System (PVFS). PVFS [6] is a parallel file system developed at Clemson University for use on Linux based Beowulf [23, 2] clusters. It is intended to serve as a platform for high performance I/O research while also meeting the need for a production parallel file system for the scientific computing community. Over time, computational capability has improved so rapidly that there is now typically a large gap between I/O performance and processing power, even in the most powerful computer systems. This has led to a situation in which file system performance is the primary bottleneck for a variety of applications. PVFS seeks to provide a solution to this problem. Many components must be brought together in order to do this, including network communications, data storage, application or system interfaces, and scheduling. This document will focus on network communications. We intend to demonstrate and analyze a system for improving network efficiency and usability for the purpose of high performance I/O. This system will take advantage of lessons learned from current file system implementations, as well as advances in cluster communication technology, to help achieve the specialized goals of parallel file systems.

Dedication

To all of my family and friends who have supported me through the years.

Acknowledgments

I would like to thank my advisor, Dr. Walt Ligon for his support and direction. I would also like to thank Dr. Rob Ross for his invaluable help.

Table of Contents

List of Figures

Figure Page

Chapter 1

Introduction

1.1 Beowulf clusters

Beowulf clusters have become increasingly popular over the last few years for parallel processing tasks [23, 2]. The beowulf architecture consists of a collection of commodity workstations connected through a dedicated local network. These systems use open source system software to provide the features expected of a parallel computer, such as message passing, process management, and global file storage space. They are intended to be used almost exclusively for parallel applications, and are thus not restricted to maintaining the functionality of individual workstations. This is a very cost effective approach to performing computationally intense research, because all of the components are either available as commodity hardware or freely distributed software.

Beowulf clusters have also provided many new avenues for systems software research. One reason for this is the diversity inherent in building custom computing machinery out of commodity parts. For example, this approach makes it more challenging to create algorithms and scheduling policies that work well in the general case, because almost no two clusters are exactly alike. Today it is no longer sufficient to just tune software for a specific vendor's hardware implementation. Assumptions made on one architecture may not be valid on another.

Systems research is also encouraged by the use of open source software. The ability to inspect, modify and redistribute modifications to even the most low level software components makes it relatively easy to contribute improvements to the community. This leads to a cycle of gradual improvement in overall Beowulf cluster technology.

1.2 Parallel file systems

One important element of Beowulf system software is the file system. This has become increasingly important as processor speeds continue to increase at a faster rate than data transfer speeds. In the past decade, CPU speed has increased from 10MHz to 1000MHz (a 100 fold increase) while average disk transfer speeds have increased from 3 Mbytes/sec to 12 Mbytes/sec (only a four fold increase) [13]. This means that applications that require large amounts of file I/O are finding disk access to be an increasingly larger bottleneck compared to computation time. In order to ease this problem, we must make more efficient use of existing storage technology. One way to do this is through the use of parallel file systems.

Parallel file systems are used to distribute data over several independent storage devices. This data is then presented to all of the application processes with a consistent name space. This combination of parallelism on both the application and storage side allows the I/O load to be distributed across an entire cluster (or at least a subset of the cluster) so that there is no single bottleneck point for file access. The bandwidth and storage capacity of this arrangement is typically much larger than what can be obtained by using a single shared file server.

One example of such a file system is the Parallel Virtual File System (PVFS) [6] which was developed at Clemson University. PVFS was designed to serve as a platform for high performance I/O research on Linux clusters. It has also grown into a stable production file system for use in the cluster computing community as a whole. PVFS makes use of common Linux features and brings them together to present a homogeneous parallel file system. In its current form it utilizes the standard Linux file system available at each node for data storage and the TCP/IP protocol for data communication between nodes.

1.3 New technologies

Since the initial PVFS design, many new technologies have become available which could potentially impact parallel file system design for Linux clusters. New hardware developments have gradually reached near-commodity status, and the collection of reliable open source system software has grown tremendously.

1.3.1 Networking

Networking infrastructure for clusters has experienced rapid development since the first Beowulf clusters were built, both in terms of hardware and software. The most common hardware available for the earliest Linux clusters was 100Mb Fast Ethernet. Since that time, Gigabit Ethernet has also become commodity hardware. Gigabit Ethernet provides much higher network bandwidth without introducing any change in application software. Several vendors have also introduced specialized cluster networking hardware, some of which can take advantage of customized network software as well. While this hardware has not reached consumer level commodity status, it is much more affordable than the custom networking hardware traditionally used in commercial supercomputers. Hardware of this category includes (but is not limited to): Myrinet hardware from Myricom, Inc. [3], SCI hardware from Dolphin Interconnect LLC [1], and Giganet hardware from Emulex Corporation [10].

There are also more choices in network software and protocols. TCP/IP has long been the standard for internet communications and was easily adopted for Linux cluster use. However, TCP/IP has disadvantages in some environments. Much of the overhead introduced in TCP/IP to handle geographically large networks and unreliable hardware simply is not necessary in a typical cluster environment [15]. Beowulf clusters, by definition, possess dedicated local networks. This type of network allows the use of much lighter weight protocols.

Alternatives for network software and protocols include the Virtual Interface Architecture [28], Score/PM [26], GAMMA [8], Active Messages [29], and vendor specific software such as GM [19]. Some of the features that may be provided by alternative networking systems such as these are:

- User level operation that bypasses the overhead of interacting with the operating system kernel during communication
- Efficient abstraction of the underlying hardware
- Lightweight transmission protocols
- Low level programming interfaces that allow developers to avoid the use of high overhead software features

Shared memory is another technology that has been around for a long time but is perhaps now easier to use on Linux clusters. Multiprocessor systems have become available at commodity prices. They allow faster interprocessor communication in some cases by using local shared memory rather than external networking hardware. At the same time, some vendors have produced products which enable the use of shared memory across a collection of nodes that would normally only communicate through traditional message passing.

1.3.2 Data storage

The most interesting technological advancements in commodity data storage recently have come from software developments. The Linux platform now has standard C library support for Posix asynchronous I/O [5], while the Linux kernel now has support for raw I/O. Asynchronous I/O allows multiple non blocking file I/O operations to be initiated and later checked for completion. This potentially allows for more efficient handling of multiple requests and the ability to overlap other application work with file I/O. Another new development is the Raw I/O interface. It is a mechanism for directly accessing disk devices at the block level without using the Linux kernel's cache and abstraction path. This opens up the possibility of writing applications that handle their own caching and device I/O independent of kernel algorithms. This can potentially boost performance of applications that have very specific I/O optimization needs that contradict generic operating system policies.

1.4 New research findings

In addition to advances in commonly available software and hardware technology, recent research and implementation has increased the knowledge base in the field of parallel I/O. There are many new ideas and implementation lessons that can now be used in parallel file system design.

1.4.1 Scheduling

One of the most important research topics explored in the first generation PVFS design is Reactive Scheduling [24]. Reactive Scheduling is a new approach to making server side scheduling decisions for parallel file systems. The main goal is to dynamically choose appropriate scheduling policies depending on the state of the system.

This is in contrast to the traditional approach of trying to optimize a single scheduling strategy to meet all of the needs of the file system.

The state of the system can be determined by system parameters, such as network or disk utilization, and also by the workload produced by the application. These parameters can be used as input to a system model. This system model then indicates what scheduling policy should be used to obtain the best performance, and dynamically switches to this policy. The scheduling policies are chosen from existing research, and could include ideas such as disk directed I/O, network directed I/O, or two phase I/O. The most important concept is the ability to correctly determine which policy is best suited to the current state.

This research has shown that scheduling decisions have an important impact on parallel I/O performance. We must be able to support efficient, modular scheduling in future work. Further work can also be done to explore policy decisions at other levels of the file system abstraction.

1.4.2 MPI-IO

MPI-IO is a standard application interface for performing parallel I/O. This standard was released as part of the MPI-2 specification in 1997 [12, 18]. It provides a portable interface for both C and Fortran applications. MPI-IO provides optimizations for discontiguous and parallel file access (through features such as derived data types and collective I/O).

MPI-IO has been widely adopted. Several implementations, such as ROMIO [27], are available. This has encouraged the creation of portable applications that take advantage of high performance I/O. The traditional portable Unix I/O interface does not provide many of the features necessary to achieve efficient parallel throughput, and the vendor specific I/O interface implementations do not work outside of their native hardware. Thus it has become important to support MPI-IO and to provide features that make it's implementation easier and more efficient.

To obtain the best performance in an MPI-IO implementation, it is helpful for a given file system to provide two features. The first is the ability to describe and service discontiguous patterns similar to those common in MPI-IO. This reduces the amount of data packing and translation that must occur outside of the file system. Secondly, the file system needs to provide an efficient, high throughput interface. MPI-IO adds a layer of abstraction which can potentially be detrimental to performance, so it is important to lower the overhead in the I/O path as much as possible to minimize this penalty.

1.4.3 Software engineering

PVFS has gradually become an accepted tool for use in production environments over the past few years. This has lead to its use in many diverse situations. This expanded use of PVFS has made it important to locate and correct software errors as quickly as possible. It has also forced the developers to continually update PVFS in order to track changes in technology.

From this we have learned the importance of thorough software engineering practice. This includes modular design, well defined interfaces, and comprehensive documentation. The use of these practices makes it much easier to support a large software project such as a parallel file system. New design decisions must acknowledge these lessons in order to be successful over the life of the project.

1.5 A new file system design

All of these changes in technology, as well as new information gained from research and implementation, have prompted the design of a new parallel file system based on

PVFS. This new file system is currently transitioning from design to implementation. It will build upon new ideas and knowledge in order to provide a more effective parallel file system for Linux clusters.

The next generation Parallel Virtual File System will be made up of several components. Some of the most important components include the network transfer mechanism, the storage transfer mechanism, the application interfaces, the server daemons, and the scheduling mechanisms. Each of these components (and several others) will be necessary in order to build a successful implementation.

1.6 Network layer requirements

This document will focus on just one component of a parallel file system for cluster computers: the network transfer layer. The network transfer layer is responsible for moving data between processes on a parallel computer. There are many ways to provide this functionality, but parallel file systems impose many requirements upon the design of such a component. The special needs of parallel I/O and the the lessons learned from current designs have prompted the following list of requirements:

- *Simple application interface*: The interface to the network transfer layer should be concise and efficient. It should well suited to describing the types of communication most often needed to perform parallel I/O, without introducing additional complexity in the design of other file system components.

- *Overlap of network I/O with other system tasks*: The network transfer layer should be designed to allow other application activity to continue as network I/O tasks are performed. This is of particular importance to server implementation, where data storage I/O can be performed simultaneously with network I/O in many cases to improve efficiency. This will become even more useful as multiprocessor systems become more common in the commodity market and thus more common in cluster applications. Multiprocessor nodes stand to benefit significantly from the ability to overlap network communication with other tasks.

- *Support for both user level and kernel level network API's*: As pointed out in section 1.3.1, there are now a variety of approaches to network communications. Some of these approaches include operating system interaction, while others perform I/O directly from the user level. It is important for implementors to be able to utilize both types of access efficiently.

- *Abstraction and modularity*: Access to the underlying network software interface should be abstracted from the user level system components. This will prevent the core design and algorithms of the parallel file system from being inextricably bound to a specific networking technology. It should be possible to modify or completely replace the underlying network technology without disturbing the implementation of other system components.

- *Efficiency*: Almost any software abstraction layer induces a performance penalty to the application. The network transfer layer should seek to minimize this penalty as much as possible. Network I/O is a performance bottleneck in many common situations, and we cannot afford to constrain it further in exchange for additional features. The file system will be sensitive to both bandwidth and latency overhead.

- *Ability to interact with multiple networks simultaneously*: If the network transfer mechanism is not bound to a single network device, then it opens up the possibility of more exotic cluster topologies. Hosts may interact with peers which exist on dissimilar networks in order to take advantage of the most efficient communication route to each.

1.7 Approach

Advances in technology and software engineering have suggested that a networking abstraction that can support a variety of network protocols would be beneficial to the development of parallel file systems. We believe that it is possible to create a network abstraction that supports multiple protocols in this manner while still maintaining high performance. The *Buffered Method Interface* (or *BMI*) has been implemented as a platform for testing the feasibility of such an interface. This implementation is intended to meet the requirements listed in section 1.6.

The remainder of this document is organized as follows. First, related work will be discussed in order to provide background for the BMI design. Next, we will outline the actual architecture of the Buffered Message Interface. Case studies of BMI implementation on top of various network protocols will be used to demonstrate its feasibility.

The results section will provide an analysis of the performance of the Buffered Message Interface. We will compare BMI performance to standard approaches to network data transfer to verify its effectiveness and determine if it meets our requirements. Finally, we will present the conclusions based on the findings of this research and propose future work.

Chapter 2

Background and related work

2.1 The Parallel Virtual File System

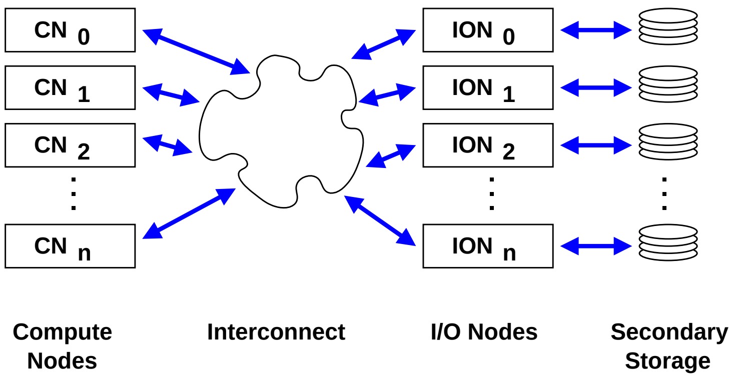
2.1.1 Motivation and goals

PVFS is a parallel file system for Linux clusters that was developed at Clemson University. It was originally designed to serve two main purposes. First of all, it is intended to be a platform for parallel I/O research. Secondly, it is intended to meet the high performance community's need for a parallel file system for Linux clusters. It has been successful in both of these goals, prompting several research papers and gaining acceptance as a high performance file system for use on production clusters.

The following is a list of some of the key goals of PVFS:

- High bandwidth for parallel read and write operations to a single file system
- Flexible application interfaces, including support from the ROMIO package for MPI-IO [27]
- Compatability with existing applications that use the native Unix I/O interface [14]
- Ability to tune file system parameters from the application level

Figure 2.1: PVFS System Overview



- Scalability
- Robustness
- Ease of installation

PVFS emphasizes performance and research viability over high availability features. It therefore does not provide software level redundancy. It also does not provide any locking mechanism within the file system itself, nor any advanced network security features such as encryption. These features are best met by other file system projects.

2.1.2 Architecture and implementation

The Parallel Virtual File System is implemented almost entirely in user space. It does not require any kernel level support for its default mode of operation. However, optional kernel level support is required to obtain compatability with applications that interact with the Unix I/O API [14]. PVFS makes use of existing technology for its most low level operations, including TCP/IP for networking and any standard Linux file system (such as EXT2 or ReiserFS) for file data storage. It is portable across 32 bit and 64 bit Linux architectures.

The basic layout of a PVFS system is shown in Figure 2.1. File data is striped across multiple cluster nodes that are connected by a local area network. No custom hardware is required since PVFS makes use of existing operating system features. Each node that possesses a portion of the file system data must run one or more local PVFS daemons that make the resources of that node available to the file system. Client applications may run on these server nodes, or they may run on separate nodes that are connected by the local area network. Client applications may access the file system through either a user level library, the ROMIO MPI-IO implementation, or the Linux kernel interface (outlined in section 2.1.4).

Manager

PVFS is made up of three primary components. The first is the manager. There is exactly one manager per file system, regardless of its size. It is responsible for maintaining metadata. In PVFS, metadata refers to the collection of properties and characteristics of files stored on the file system. This includes information such as ownership and permissions that are not actually part of the file data.

In addition to the standard Unix file properties such as those listed above, the manager also maintains metadata that is unique to PVFS. This includes the physical distribution information for each file. This is used to determine which I/O servers in the file system possess file data, and how the file data is distributed among them.

The PVFS manager serializes all metadata operations from clients and ensures metadata consistency. This is made easier by the fact that there is only one manager, eliminating the need to maintain synchronization with another node for important metadata information. It may seem like a performance bottleneck to rely on a single manager, but this is eased by the fact that the manager does not participate in file I/O operations. Once a client has verified permissions and metadata information, it does not communicate with the manager when reading or writing file data. This is handled by the PVFS I/O daemons, which will be outlined shortly.

I/O daemon

The I/O daemon is responsible for servicing I/O requests and storing file data. There may be any number of I/O daemons, from as few as one to as many as the scalability of the system will permit. Each I/O daemon stores data on the local file system of the node that it is running on. When multiple I/O daemons are being used, the data is striped across them in round robin fashion. The stripe size, offset, and number of I/O daemons to use can be specified by the user on a per file basis.

The use of multiple I/O daemons introduces parallelism on the server side of the file system. A client may have parts of its request serviced from several different servers, thus leading to utilization of several separate disks and network connections simultaneously, rather than waiting for service at one particular bottleneck point. This style of access is tailored to improving throughput for parallel applications, especially those which demand large amounts of I/O. It may not be optimal for non-parallel applications, because the client becomes a bottleneck for the flow of data, thus negating the advantage gained by having servers operate in parallel.

Each I/O daemon operates independently of other I/O daemons in the system. It is only aware of the portions of a file that it is in control of at any given time. File distribution remains static over the lifetime of a file in PVFS.

Client library

The PVFS client library enables applications to interact with the file system. It is a C library for use in user level programs, and does not require any interaction with the Linux kernel for communication. It provides a native PVFS API, which is derived from the standard Unix I/O API. Among other things, it includes functions for opening, closing, reading, and writing to PVFS files. It also adds the ability to specify PVFS specific parameters for files, such as physical stripe size and number of I/O daemons to use.

The native PVFS library also provides the ability to make discontiguous I/O requests. This is an important feature for parallel applications that is not provided in standard Unix interfaces. This is done by using a *partitioned file interface*. File partitioning allows a process to alter its view of a file logical file so that it can access discontiguous regions with single read or write requests. It is similar to Vesta's logical file partitioning [9] and to file views provided by MPI-IO [12, 18].

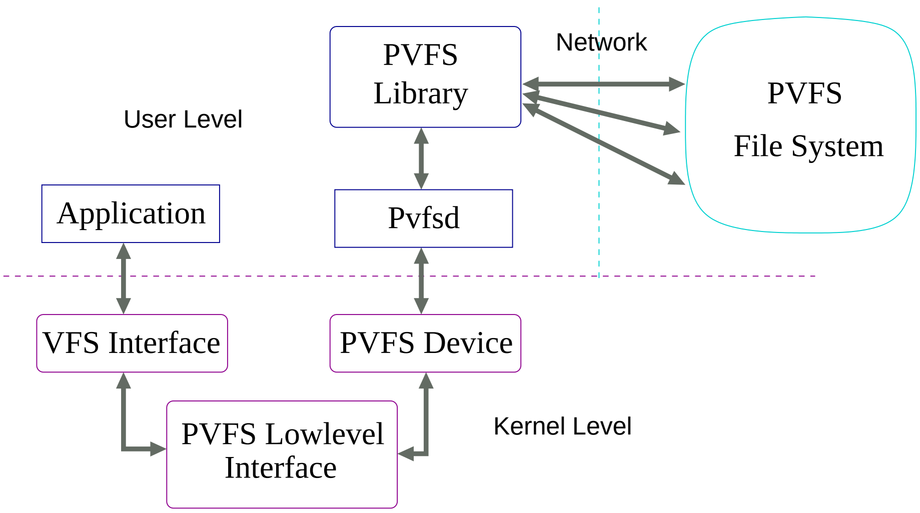
The PVFS library is responsible for orchestrating communication with the manager and any I/O daemons as necessary. For large systems, this may involve communicating with hundreds of servers. All of this is hidden from the application.

2.1.3 Low level I/O

All PVFS data is stored on standard local file systems. Each I/O daemon stores its portion of the data files on a local file system, and the manager stores metadata information on its local file system as well. This is in contrast to other file system implementations which write raw data to disks. By avoiding the use of raw disk blocks, complexity is reduced, and PVFS benefits from features of the local file system such as caching or journaling.

All inter-node communication is carried out using the standard TCP/IP protocol. This was the primary communication mechanism available for Linux clusters when PVFS was first designed. It provides reliable, ordered delivery and flow control for data communications. It is also available on almost every Linux cluster.

Figure 2.2: PVFS kernel architecture



2.1.4 Unix I/O compatability

The PVFS client library provides an optimized path for applications to interact with PVFS. It also provides access to PVFS specific parameters. However, it is not compatible with existing applications. This prompted the design of the PVFS kernel package as an alternative. It provides a kernel path for PVFS so that applications can interact with it just as they would any other file system. When using this package, PVFS is mostly indistinguishable from a more traditional file system from the user's point of view.

The architecture of the PVFS kernel implementation is shown in Figure 2.2. It consists of both a user level and kernel level component. The kernel level components are implemented as a module, while the user level components are implemented in a client side daemon known as the pvfsd. The pvfsd is responsible for actually translating file requests to native PVFS requests and communicating them over the network to the PVFS file system. This is done at user level because it provides more flexibility than a full kernel implementation. It allows the client to utilize any network mechanism that is required (some of which may not be available within the operating system kernel). It also allows the use of the standard PVFS library for handling requests, rather than maintaining an independent interface within the kernel.

Communication between the user level pvfsd process and the kernel module is carried out through use of a special device file. Requests are read out of the device file by the pvfsd, and responses are written back into it. Several methods of transferring bulk data through this interface are provided, but their operation is beyond the scope of this document.

This implementation attempts to be as modular and portable as possible, so that it can survive multiple generations of file system design. All code specific to the PVFS file system implementation is maintained within a strict interface.

2.2 Virtual Interface Architecture

2.2.1 User level networking background

Communications hardware has advanced rapidly in recent years, creating a variety of alternatives for high performance communications. Most Linux clusters were originally constructed with 10 Mbit or 100 Mbit Ethernet, but such networks have recently been augmented with more advanced commodity options, such as Gigabit Ethernet, as well as specialized system area networks, such as Myrinet [3].

These advances in hardware have prompted research into the software aspect of communication as well. Cluster specific system area networks, in particular, may benefit from software interfaces and protocols which take advantages of assumptions that do not apply to general case local area networks. There are several aspects of this level of communication that may be targeted for improvement. Two notable weaknesses in the use of general purpose communications in clusters are in the level of protocol complexity and the amount of operating system interaction that takes place during data transfer [4]. Protocol complexity arises from multi-layer software implementations that provide a wide abstraction from hardware, as well as from the inclusion of a large suite of features which may or may not be utilized. Operating system overhead arises from relying on the system kernel to provide memory protection and resource allocation. These issues can be addressed by designing messaging systems that more closely match the ability of the underlying hardware, and also by removing the kernel as much as possible from the critical path of data communication.

This broad approach to optimizing application interaction with the network has been termed *user level networking*. This approach was largely pioneered by the Active Messages project [29], which also encompassed several other topics within application and network interaction. This work continues today and has progressed beyond research to include specific vendor offerings, such as GM [19], as well as cooperative industry initiatives such as VIA [28].

2.2.2 VIA specification

The Virtual Interface Architecture is an attempt at standardizing both the semantics and interface to user level networking across a variety of hardware and software vendors and implementors. It officially came into being in 1997 with the release of the Virtual Interface Architecture Specification, which was the result of a collaboration being several leading vendors, including Microsoft, Compaq, and Intel [28]. This document outlines the architecture of VIA from both an implementation and usage point of view. It draws heavily from well known research efforts such as Active Messages, but its aim is to provide a usable, production level standard for user level networking on system area networks.

2.2.3 Architectural overview

The Virtual Interface Architecture model consists of several components, including the *VI Provider*, *VI Consumer*, *Virtual Interfaces*, and *Completion Queues*.

The VI Provider is made up of the hardware and software components that interact to provide the resources necessary for a virtual interface. These resources include memory protection, connection setup and teardown, and error management. The hardware utilized is generally a network interface card. The hardware may be specifically designed to support VIA (and is thus considered “VIA aware”) or it may be a more traditional design, for which certain features of the VIA architecture must be emulated. The software component of the VI Provider is usually a kernel level driver. Note that this kernel driver is only responsible for a limited number of resources, and is typically only invoked during initial setup of communication between two hosts. It is not directly involved in the data path for communication.

The VI Consumer can roughly be thought of as the user of the VI resources. This is of course includes the application, but also encompasses the interface that the application uses to interact with the other VI components. This level of the interface is normally implemented as a user level library, and it has a strictly defined set of functions that are used to invoke communication mechanisms. These functions trigger the necessary kernel level setup mechanisms and abstract the complexity of the other components.

The Virtual Interface itself is the mechanism that allows the consumer to directly interact with with the provider in order to actually transfer data. There is one Virtual Interface per peer that the host wishes to communicate with. It includes work queues for both send and receive operations. These work queues are a key concept in understanding how VIA operates. The queues do not actually contain the data to be transfered in communication operations. Instead, they contain compact *descriptors* that describe the data to transfered, as well as other details (such as status information) that are necessary to describe the communication. When the VI posts descriptors to the work queues, it uses *doorbells* to indicate to the network adapter that new work is available. Doorbells are very small, simple triggers that are intended to be implemented directly from the hardware so that no operating system intervention is required to notify the hardware. The doorbells may be emulated if needed, however. The goal is to provide information to the network hardware as efficiently as possible so that it can use hardware mechanisms such as direct DMA transfers to move application data, rather than relying on additional kernel level buffers for this purpose.

Completion Queues are provided as a mechanism for the VI to notify the Consumer that messages have been completed. Once descriptors have been processed from the work queues, their status is filled in to indicate success or failure. They may then be transfered to completion queues that were specified by the Consumer so that the application can be made aware of the completion. Again, the goal is to indicate status changes and to transfer information about communication without invoking kernel context switches or unnecessary interrupts.

2.2.4 Usage

The VIA model for communication implies that the application has more responsibilities in the communication process than would be expected of traditional software interfaces. First of all, the application must be able to create and manage the descriptor and doorbell structures that are necessary for initiating data transfers. It must explicitly post these structures and inspect them upon completion.

Additional constraints are also placed upon the memory regions that the application may use for communication. Networking hardware normally requires message data to be locked into physical memory and specified in terms of physical, rather than virtual addresses before being sent across the network. Traditional network protocol implementations allow the operating system driver to handle this requirement rather than expose it to the user. VIA, however, requires that the user explicitly perform this memory registration before submitting a descriptor that references the region.

Only memory that has been registered with the VI Provider may be transfered. This advance registration allows the Provider to directly access the region without utilizing an intermediate buffer. These regions may also be reused, so that the overhead of registration need not be incurred for every message.

The communication semantics of VIA are very lightweight. For example, no message buffering is provided for receive operations. This means that all receive buffers must be posted before the message data arrives, or else it may be discarded. This introduces more management responsibilities on the part of the application to ensure that buffers are provided in a timely manner. In addition, the VIA specification outlines three levels of reliability which may optionally be provided. These levels of reliability differ in the amount of assurance that the application is given concerning the successful arrival of messages. If a VIA implementation does not provide a high enough level or reliability for the needs of an application, it may require another layer of software in order to provide this functionality.

Due to the above application implementation requirements, VIA is most often used as a foundation for a higher level API, such as MPI [18], or for use in implementation of system software. These environments can take advantage of the efficiency of the Virtual Interface Architecture without requiring a new learning curve for the applications programmer.

2.2.5 Implementations

There have been several adopters of the VIA specification thus far. A few notable ones are listed below with a brief outline of the approach taken to providing the VIA features. This is a sampling of the level of acceptance within the high performance computing community.

Berkeley VIA

The Berkeley VIA project (based at UC Berkeley) is a research oriented project that has provided cross platform VIA implementations for Myricom's Myrinet hardware [3]. They seek to provide a high quality VIA implementation, explore its performance characteristics, and investigate possible improvements to the architecture. It is not a full implementation of the specification, but it provides enough functionality to evaluate performance and support most VIA applications. The Myrinet hardware that it utilizes, though not designed explicitly for use with VIA, is highly programmable and provides most of the hardware features suggested for VIA implementation.

M-VIA

M-VIA is a research prototype VIA implementation from the National Energy Research Scientific Computing Center [16]. Its primary features include a modular design which should ease the work of porting to new hardware. It provides a full implementation of the VIA standard and also allows coexistence with other protocols which may be supported by target hardware devices. M-VIA currently supports a variety of Ethernet hardware by emulating in software some of the necessary device features, though it can also take advantage of VIA accelerated hardware. Future releases target a more ambitious range of networking hardware.

Giganet

Giganet is a vendor hardware offering from Emulex Corporation [10]. It includes a full VIA software implementation, and network interface cards that are designed with VIA support in mind.

2.3 Virtual Machine Interface

The Virtual Machine Interface [21] is a high performance messaging API developed by The University of Illinois, Urbana-Champaign and the National Center for Supercomputing Applications. It provides a uniform interface for interacting with dissimilar networking systems. This interface may be used directly by an application, or used as the foundation for implementing a higher level interface such as MPI[18]. VMI provides connectionless, reliable, ordered delivery. It also requires communication buffers to be registered, much like VIA [28].

VMI also provides several other advanced features that make it stand out among other network abstraction implementations:

- Different network types are supported through the use of dynamically loadable modules. This means that applications do not need to be recompiled in order to take advantage of new devices, nor to acomidate changes in cluster architecture.

- VMI provides collective notification of process failure. If a single process in a computation crashes, then all peer processes are notified. This accommodates graceful termination of the computation as a whole.

- VMI is intended to be portable across dissimilar platforms. It has been tested with both Linux and Microsoft Windows cluster environments, and allows both to interact on the same computation.

- A single host is allowed to communicate over multiple devices simultaneously. This allows VMI to be used in heterogeneous network environments. Furthermore, extensions to VMI allow it to serve as a bridge across dissimilar networks, thus allowing full interprocess communication when only a limited number of nodes share connections to both networking systems.

As of this writing, VMI supports shared memory, VIA, and TCP/IP socket communications. Support is planned for Myrinet and SCI networks as well.

2.3.1 Shared memory

The VMI shared memory device is used between processes that are located on the same physical node within a cluster. It operates by providing a unique shared buffer and synchronization structure between each pair of communicating processes. The shared buffer consists of a contiguous 1 Mbyte region that is writable by one process and read only for another. It may be allocated in 1 Kbyte pages. Synchronization is handled through a bounded circular queue that indicates the send window between a pair of nodes. Lock based synchronization is avoided because the buffer is only writable by one process at any given time.

2.3.2 VIA

The VIA device has been implemented using the Giganet device driver. VIA already provides many of the primitives required for VMI module implementation. However, it does not provide flow control. VMI therefore implements a credit based flow control mechanism on top of VIA within the module.

2.3.3 TCP/IP sockets

The TCP/IP socket interface is not as close of a match to the VMI module requirements as VIA or shared memory communications are. Thus implementation for this protocol is more difficult. VMI currently supports a proof of concept TCP/IP module that has not yet been optimized for performance.

2.4 Message Passing Interface

The Message Passing Interface is a specification for application level message passing. It was defined by the MPI Forum in an attempt to provide a standard to ensure portability between parallel computers from a variety of vendors and research groups. Prior to the drafting of the MPI specification, many vendors provided their own unique libraries for message passing which made it difficult to create portable applications. Many of these libraries shared the same fundamental features but differed widely in terms of interfaces and syntax.

The initial MPI Standard (Version 1.0) was completed in May of 1994. Work on this standard was later continued, resulting in the 1.1, 1.2, and 2.0 MPI Standards [17].

The MPI Standard encompasses both the application interfaces and semantics of the message passing system. This system provides many features, including point to point as well as collective communication. It also provides other services such as consistent process naming, virtual topologies, heterogeneous data conversion, performance monitoring tools, and profiling interfaces. The MPI-2 Standard brought enhancements such as parallel I/O, remote memory operations, and dynamic process management.

MPI has quickly become the default message passing library for parallel applications. There are implementations available for every major modern architecture.

2.4.1 MPICH

MPICH (or MPI Chameleon) is a portable implementation of the MPI standard supported by Argonne National Laboratory [11]. MPICH is unique in that its development began while the initial standard was still being drafted. This provided the MPI Forum with immediate feedback from a design that tracked the standard as it was developed. This also resulted in the availability of a working MPI implementation as soon as the standard was finalized, which aided in its acceptance.

MPICH was originally constructed by taking advantage of existing code from preceding systems. One of these was parallel programming library known as P4, which provided portable shared memory and message passing components. Another was Chameleon, a package which focused on portability over a variety of message passing architectures. The final precursor was zipcode, which provided concepts for scalable libraries, such as contexts and groups.

The two most important design goals of MPICH were portability and efficiency. MPICH runs on a variety of systems and provides low level interfaces for implementers to quickly port it to other environments. However, it strives to do this without sacrificing overall performance.

MPICH architecture

The MPICH architecture was carefully designed to meet the goals of efficiency and portability. It can best be described in terms of three major components:

- *High level code*: The highest level MPICH code includes many concepts (such as groups, communicators, and opaque objects) which are independent of the communications mechanism. Therefore, this portion of the design is provided as a portable implementation that can be expressed in terms of lower level abstractions.

- *Abstract Device Interface*: All high level MPICH code is written on top of the Abstract Device Interface (ADI). There are many separate implementations of the ADI for different architectures. It provides an interface for implementers to quickly integrate new architectures without having to rewrite the full MPI library from scratch.

- *Channel Interface*: One implementation of the ADI uses the channel interface. The channel interface is a very small abstraction of the communications mechanism which can be implemented with as few as five functions. This provides the quickest path to implement a new device, but at the cost of efficiency.

The idea is for implementors to rapidly prototype a new device by implemented support for it at the channel interface level. As the implementation progresses, higher levels of abstraction can be replaced by device specific code, so that the device eventually has its own ADI implementation and perhaps even optimizations of higher level functionality. Implementors therefore have the advantage of a portable implementation for rapid prototyping but are not constrained by it in the long term.

2.5 Bringing together related work

Three distinct messaging abstractions have been discussed in the previous section: the Virtual Interface Architecture, the Virtual Machine Interface, and the Message Passing Interface. These tools are all currently available and perform their respective tasks quite well. However, the realm of parallel I/O on Linux clusters has specific needs which are not yet met by any single message passing implementation.

In order to be successful within the field of parallel I/O, a network abstraction needs to bring together a hybrid of several features. It must be robust in the face of small network failures so that errors on individual hosts do not impact system wide activity. It furthermore must be capable of sustained client/server style operation. Unexpected hosts may connect and disconnect from a file system many times while the system is running.

A network abstraction for parallel I/O must also be efficient for the problem domain that it is solving. There are many features critical to general purpose messaging tools that simply are not applicable in a client/server based system level toolkit.

Supporting unnecessary features is almost undoubtably a source of overhead and maintenance difficulties.

Finally, the network abstraction must share the file system's ability to work efficiently and reliably on a variety of existing cluster architectures. Many production clusters can not afford to experiment with sweeping system level changes at the risk of disrupting ongoing computational work. We want to support these traditional systems while also leaving the door open for work with more exotic systems when possible.

Chapter 3

Design of the network transfer layer

The Buffered Message Interface (BMI) has been designed to serve as the network transfer layer for a next generation parallel file system. It is implemented as a library that provides a standard interface for communication between system software components. Although designed for use within the Parallel Virtual File System, BMI is an independent entity which may be useful in other environments as well.

3.1 Communications model

BMI is a message passing system that provides reliability, ordering, and flow control. If a particular underlying network protocol does not provide one of these features, then BMI is responsible for implementing it.

All communications operations in BMI are nonblocking. In order to send a message, the user must first *post* the message to the interface, then *test* it for completion. The same holds for receiving messages. Once testing indicates that a message has completed, the user must check the status of the message in order to determine if it completed successfully or not. Partial completion is not allowed.

In fact, every function defined as part of the BMI interface is nonblocking. Each function may perform work before completing, but this work is guaranteed to complete within a bounded amount of time. This restriction implies that it may be necessary to test for completion of a message several times before it actually completes. There is no mechanism that allows the interface to "wait" indefinitely for completion of a particular operation. This design decision was made because blocking network calls (especially in large parallel systems) are prone to problems with robustness and scalability. They may cause an application to hang in the event of network or programming errors. This is not acceptable within low level system services.

When posting receive operations, the user must specify the address of the sending host and the size of the message to accept. The user cannot post receives that match wildcard addresses. The only exceptions to this rule are unexpected messages, as defined in section 3.3.

BMI is a connectionless interface; the user does not have to establish or maintain any link between hosts before sending messages. The BMI implementation may maintain connections internally if needed for a particular network device, but such details are not exposed to the user.

3.2 Memory buffers

The user must specify a memory buffer to use when posting send and receive operations. This buffer may be a normal memory region, or it may be a buffer that was allocated using BMI memory management functions. If the user elects to allocate the memory using the BMI facilities, then BMI has the opportunity to optimize the buffer for the type of network being used. This mode of operation is preferred for achieving optimal performance. However, normal memory buffers are also allowed in order to better support certain scenarios common to file system operations. Some file system operations act upon existing memory regions (for example, the client side Unix read() system call). In these situations, we would like to avoid imposing a buffer copy, and instead give the BMI layer the flexibility to handle the buffer at a lower level if possible.

If a memory buffer is allocated using BMI function calls, then it must also be deallocated using BMI. These buffers are not guaranteed to be manageable by standard operating system libraries.

3.3 Tags and unexpected messages

The BMI interface allows the user to specify a *tag* for each message. A send operation with a specific tag may only be accepted by a receive operation that specifies a matching tag. This therefore provides for the user a mechanism to differentiate between distinct classes of messages. However, one particular tag is reserved by the interface to have special meaning. This tag marks a message as *unexpected*. Unexpected messages are messages that are sent without the receiving host explicitly requesting the communication. In other words, the receiving host does not post a matching receive for this type of message. Instead, it must periodically check to see if any unexpected messages have arrived in order to receive them successfully. This is the equivalent of "listening" for new requests in a more traditional networking system. Unexpected messages may come from any host on the network. Communication between two hosts is typically initiated by one of the hosts sending an unexpected message to the other.

3.4 Client/Server paradigm

The BMI system is better suited for client/server application models than peer to peer models. This is made evident by the concept of unexpected messages as defined

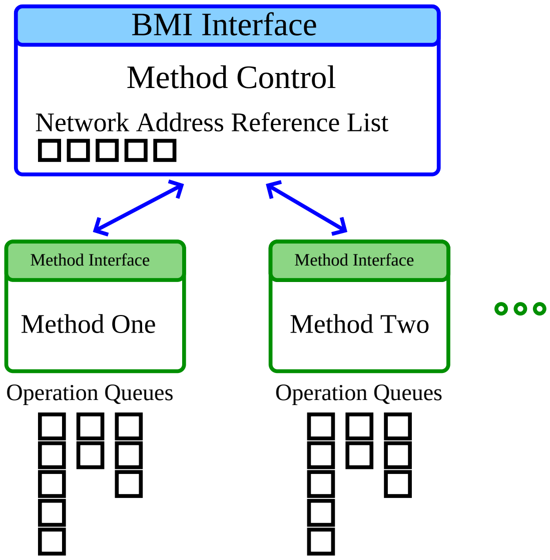


Figure 3.1: BMI Architecture

above. Consider the simple example of communication between two hosts. Typically only one of the hosts will look for unexpected messages. This is the "server". The other host acts as a "client" by sending unexpected messages to the server that prompt it to perform some service. This service may involve the exchange of further messages between the two hosts.

3.5 Architecture

The overall architecture of BMI is shown in Figure 3.1. Support for individual network protocols is provided by BMI *methods*. There may be any number of methods active at a given time. This collection of methods is managed by the *method control layer*. The method control layer is also responsible for presenting the top level BMI interface to the application.

3.6 Method control

From a high level, the method control layer is responsible for orchestrating network operations and managing the network methods. This includes several responsibilities, including address resolution, method multiplexing, and providing a stable BMI user interface. It also provides a library of support functions that may be useful to method implementors.

One of the most important tasks of the method control layer is the multiplexing of network methods. When an operation is posted by the user, it is up to the method control to decide which method will service the operation. Likewise, when the user tests for completion, the method control must test the appropriate methods for the operations of interest.

The method control layer provides the BMI user interface. This is the API used by applications that communicate using BMI. The BMI interface functions are converted into the appropriate low level method requests that are needed to complete operations.

Address resolution is the final major responsibility of the method control. The method control manages the BMI level addresses and makes sure that the name space is consistent to the user, regardless of which methods are in use. It does so by maintaining an internal *reference list* for addresses. Each network address has a unique reference that provides mappings between BMI user level addresses, the string representation of addresses, and the method specific representation of addresses. The BMI user level addresses are handles for network hosts that the application uses when calling BMI functions. The string representation is the ASCII host name of the hosts before they are resolved by BMI (as read from a "hosts" file, for example). Finally, the method address is the representation that that methods use for identifying hosts, which may contain information specific to that particular protocol. Note that method addresses are never, under any circumstances, exposed to the application. They are reserved for internal BMI use only.

3.7 Methods

Each method is implemented as a dynamically loadable module. This module must provide (and strictly adhere to) a predefined *method interface*. It supports reliable, ordered delivery and flow control for the protocol that it controls. Aside from meeting these semantics and adhering to the method interface, there are no other restrictions on how the method should be implemented. Support libraries are provided for certain features that are common to many methods, but their use is optional.

Each method is responsible for maintaining the collection of operations that it is working on, usually through operation queues. These collections of operations are private to each method.

3.8 BMI user interface

The BMI interface can be separated into four small categories of functions: message initiation, message testing, memory management, and utilities.

The message initiation functions are used by an application to request the sending or receiving of network buffers:

- **BMI_post_send(id, destination, buffer, size, buffer_flags, message_tag)**: Posts a send operation from the specified buffer. The id is written in by the function and serves as a unique handle for the operation to be used when testing for completion. The buffer_flags are used to indicate whether the buffer was allocated by the application or by BMI.

- **BMI_post_recv(id, source, buffer, size, buffer_flags, message_tag)**: Posts a receive operation. The argument semantics are the same as those used in BMI_post_send().

- **BMI_unpost(id)**: Forcefully aborts a previously posted operation. The status of the target operation should still be retrieved through the use of a BMI_test function(), however. It may have completed successfully before the BMI_unpost() was processed.

- **BMI_addr_lookup(new_addr, id_string)**: Performs a lookup on the textual representation of the host address specified by id_string. The resulting BMI specific address handle is filled into the new_addr parameter and may be used for subsequent network initiation functions.

The message testing functions are used to check for completion of network operations:

- **BMI_test(id, outcount, state)**: Tests for completion of a single network operation, as specified by the id argument. Outcount indicates how many operations completed (which will either be zero or one in this case). The state parameter is filled in with the state of the operation in question once it completes.

- **BMI_testsome(incount, id_array, outcount, index_array, state_array)**: Tests for completion of any of a specified set of operations. The set of operations to look for is specified by an array of id's of size incount. Outcount is filled in to indicate how many of the target operations completed, while index_array and state_array indicate exactly which operations completed and what their final state was. BMI_testsome() ignores any id's within the id_array that have been set to the null value of zero.

- **BMI_testglobal(incount, id_array, outcount, state_array)**: Tests for completion of *any* operations that are currently in progress. Incount specifies how many operations the caller is willing to accept with one invocation of

BMI_testglobal. Id_array, outcount, and state_array are filled in to indicate which operations completed and what their final state was.

- **BMI_testunexpected(incount, outcount, info_array)**: Tests for completion of any newly arrived unexpected messages. The incount indicates how many operations the caller is willing to accept, while the outcount indicates how many actually completed. The info_array is filled in with a description of each completed operation, including the source address, buffer location, and size. These parameters are not known in advance by the caller (hence the *unexpected* nomenclature).

The BMI memory management functions are used to control memory buffers that are optimized for use with BMI:

- **BMI_memalloc(address, size, send_recv_flag)**: Allocates an optimized memory buffer of the requested size. The address parameter indicates the remote host that will participate in the transmission of the buffer. The send/recv flag indicates whether the buffer will be sent or received from the local host.

- **BMI_memfree(address, buffer, send_recv)**: Frees a memory buffer that was allocated with BMI_memalloc(). The address and send_recv parameters possess the same semantics as those used in BMI_memalloc().

The final collection of functions perform various utility tasks that are not directly involved in network I/O:

- **BMI_initialize(module_string, listen_addr, flags)**: Initializes the BMI system. This function must be called before any other BMI interface functions. The module string is a comma separated list of dynamic method modules to use. The listen addr is a comma separated list of parameters that the network methods use for receiving messages (if needed).

- **BMI_finalize()**: Shuts down the BMI library. This function should only be called once all network communication is completed. It will forcefully terminate any outstanding operations.

- **BMI_set_info(address, option, parameter)**: Sets optional BMI parameters. If the address is specified, the function will on affect the method that is responsible for that address. Otherwise, the function has a global impact on the BMI methods.

- **BMI_get_info(address, option, parameter)**: Queries BMI for optional parameters.

3.9 Immediate completion

The default model for each network operation is to first post it and then test for completion. However, there are often instances in which operations can complete immediately (during the post procedure) and thus do not require the extra test step. Examples of this occur when TCP sockets buffers are large enough to allow a message to be sent in one step without blocking. This may also occur on the receive side of communications if the required data has already been buffered by the BMI library when the receive operation is posted.

In these situations, it would be good to avoid the overhead of needlessly calling the test function. We therefore allow *immediate completion* from any post function. Immediate completion is indicated from post functions by a return value of one. BMI library users should always check this return value so that they are aware of opportunities to skip the test phase of communication.

3.10 Method interface

The method interface is very similar to the BMI user interface. It implements roughly the same functions. However, it includes minor variations that take into account the fact that operations at this level are targeted for a single specific method.

The following listing describes the BMI method interface. Note that the address arguments in this interface (*source*, *destination*, and *new_addr*) are of the special *method address* structure type. Each method address contains binary format information that can only be understood by the specific method that created it.

- **BMI_method_post_send(id, destination, buffer, size, buffer_flags, message_tag)**: Posts a send operation from the specified buffer. The id is written in by the function and serves as a unique handle for the operation to be used when testing for completion. The buffer_flags are used to indicate whether the buffer was allocated by the application or by BMI.

- **BMI_method_post_recv(id, source, buffer, size, buffer_flags, message_tag)**: Posts a receive operation. The argument semantics are the same as those used in BMI_method_post_send().

- **BMI_method_unpost(id)**: Forcefully aborts a previously posted operation. The status of the target operation should still be retrieved through the use of a BMI_method_test() function, however. It may have completed successfully before the BMI_method_unpost() was processed.

- **BMI_method_addr_lookup(id_string)**: Performs a lookup on the textual representation of the host address specified by id_string. The return value is the resulting method address structure as generated by the method to represent the address.

- **BMI_method_test(id, outcount, state)**: Tests for completion of a single network operation, as specified by the id argument. Outcount indicates how many operations completed (which will either be zero or one in this case). The state parameter is filled in with the state of the operation in question once it completes.

- **BMI_method_testsome(incount, id_array, outcount, index_array, state_array)**: Tests for completion of any of a specified set of operations. The set of operations to look for is specified by an array of id's of size incount. Outcount is filled in to indicate how many of the target operations completed, while index_array and state_array indicate exactly which operations completed and what their final state was.

- **BMI_method_testglobal(incount, id_array, outcount, state_array)**: Tests for completion of *any* operations that are currently in progress within a single method. Incount specifies how many operations the caller is willing to accept with one invocation of BMI_method_testglobal. Id_array, outcount, and state_array are filled in to indicate which operations completed and what their final state was.

- **BMI_method_testunexpected(incount, outcount, method_unexpected_info_array)**: Tests for completion of any newly arrived unexpected messages. The incount indicates how many operations the caller is willing to accept, while the outcount indicates how many actually completed. The method_unexpected_info_array is filled in with a description of each completed operation. Note that this structure is different from the info_array argument to the top level BMI_testunexpected() function. This is because it contains information that is private to the BMI library and therefore should not be visible to a BMI user.

- **BMI_method_memalloc(size, send_recv_flag)**: Allocates an optimized memory buffer of the requested size. The send/recv flag indicates whether the buffer will be sent or received from the local host. No address argument is given because the top level interface has already used that data to determine which method to use. It is not needed at this level.

- **BMI_method_memfree(buffer, send_recv)**: Frees a memory buffer that was allocated with BMI_memalloc(). The send_recv parameter has the same semantics as in BMI_method_memalloc().

- **BMI_method_initialize(listen_addr, method_id, flags)**: Initializes the method. This function must be called before any operations are processed by the method. The listen_addr is a method address that contains information about how the method should listen for new messages. The method_id parameter is used to inform the method of the id handle that will be used to reference the method and it's address structures. This is assigned by the method control code to prevent collisions between method identifiers.

- **BMI_method_finalize()**: Shuts down the method. This function should only be called once all network communication is completed. It will forcefully terminate any outstanding operations. Each individual method can be shut down independently without disrupting any other operations.

- **BMI_method_set_info(address, option, parameter)**: Sets optional method specific parameters.

- **BMI_method_get_info(address, option, parameter)**: Queries the method for optional parameters.

3.11 Support libraries

The BMI library provides several support functions which may aid method programmers when implementing support for new protocols. Each method can expect these functions to be visible to it once it has been dynamically loaded into the library. These functions are intended to be as generic as possible so that they may be used by a variety of different methods.

3.11.1 Operation queues

Every prototype method implemented so far makes use of FIFO queues to keep track of pending operations. Operations are described by generic operation structures that include common parameters (such as buffer size and location). This structure also includes abstract storage space for private method specific parameters (such as flow control or device management information). The operation queue mechanism in BMI is based heavily on the doubly linked list implementation found in the 2.2 and 2.4 series Linux kernels. This implementation is used throughout the kernel in areas such as CPU scheduling and the TCP/IP stack which require data structures to be optimized for speed.

- **op_queue_new()**: Creates a new operation queue.
- **op_queue_cleanup(old_op_queue)**: Destroys an existing operation queue as well as any operations contained within it.
- **op_queue_add(target_op_queue, method_op)**: Adds a method operation onto the tail of a queue.
- **op_queue_remove(method_op)**: Removes a specific operation from the queue in which it resides.

- **op_queue_search(target_op_queue, key)**: Searches for an operation that matches the characteristics specified by the key. All searches begin at the head of the target operation queue.

- **op_queue_empty(target_op_queue)**: Determines whether a queue is empty or not.

- **op_queue_count(target_op_queue)**: Counts the number of entries within an operation queue. This function requires iteration through every element of the queue. It is therefore only suitable for debugging purposes in which performance is not critical.

- **op_queue_dump(target_op_queue)**: Prints out information about every operation in the queue. Only used for debugging and prototyping purposes.

Two related functions are also provided for managing the creation of operation structures:

- **alloc_method_op(payload_size)**: Allocates a new operation structure, including enough room for the private data payload that a particular method may wish to store within it. Note that this private data is provided in a region contiguous to the generic structure for efficiency.

- **dealloc_method_op(target_op)**: Deallocates an existing method operation.

3.11.2 Method address support

Method address structures are used by methods to identify network hosts. Like operation structures, they contain private storage for internal method use. Three functions are provided to aid in managing these structures:

- **alloc_method_addr(method_id, payload_size)**: Creates a new address structure. The method_id field is used to tag the structure as belonging to the method that created it. The payload_size indicates how much space should be set aside within the structure for private use by the method.

- **dealloc_method_addr(old_method_addr)**: Destroys an existing method address structure.

- **bmi_method_addr_reg_callback(target_method_addr)**: This is called by a method to inform the method control layer that it should register a new method address structure. The function is typically invoked when an unexpected message arrives and the method must autonomously create a new address structure to represent the source host. The new method address structure must be registered with the method control layer so that it is aware of the new structure for bookkeeping purposes.

3.11.3 Logging and debugging

The BMI library includes a set of functions known as the *gossip* library which may be used for reporting errors, logging messages, or providing debugging information. The gossip library was created to provide a consistent interface for performing these tasks. It also can be implemented with various backends that can be switched at runtime to control where the log messages are actually recorded. As of this writing it supports stderr, syslog, and file based logging. In the future it will support other mechanisms such as in core ring buffers.

Gossip also supports setting debugging masks, which can control which classes of messages are actually recorded. This is useful in BMI for selecting which methods actually display debugging output.

- **gossip_set_debug_mask(debug_on, mask)**: Controls whether debugging messages are on or off. The mask parameter specifies what class of debugging messages will be displayed if debugging is turned on. Note that critical error messages *cannot* be disabled.

- **gossip_enable_syslog(priority)**: Enables the syslog logging facility and sets the syslog priority that will be assigned to each message.

- **gossip_enable_stderr()**: Enables the printing of error messages to stderr.

- **gossip_enable_file(filename, mode)**: Enables the logging of error messages to a specific file. The mode parameter is useful for specifying if the file should be truncated or appended.

- **gossip_disable()**: Turns off the gossip library.

- **gossip_debug(level, format, ...)**: Logs a printf() style message with the specified debugging level.

- **gossip_err(format, ...)**: Logs a printf() style error message. This message will be recorded regardless of the current debugging mask.

- **gossip_ldebug(level, format, ...)**: Same as the gossip_debug() function, except that it also displays the file name and line number from which the message originated on systems with preprocessors that support this feature.

- **gossip_lerr(format, ...)**: Same as the gossip_err() function, except that it also displays the file name and line number from which the message originated on systems with preprocessors that support this feature.

3.11.4 Operation id's

Each method is responsible for creating opaque id's that can be used to refer to operations that are currently in progress. Typically these id's will be used to map user requests to specific operation structures. The *id_generator* library is available to aid methods in performing this mapping operation. This functionality is provided within a discrete interface to allow for multiple implementations which may use hash tables or other data structures to store mapping information. It also insures that the id space is consistent across all methods.

- **id_gen_fast_register(new_id, void* item)**: Registers a new structure with the interface and creates a new id that may be used to reference it.
- **id_gen_fast_lookup(id)**: Returns a pointer to the original data structure that was associated with the given id.

All of the interfaces listed in the preceding section come together to form the *method support libraries* that method implementors should take advantage of when creating new methods.

3.12 Method control implementation

The method control layer (as introduced in section 3.6) is responsible for performing conversions between the BMI user interface and the method interface, including any method multiplexing that this implies. This is a relatively thin layer of software, but it plays a critical role in providing core BMI features. Some of the more interesting tasks that it performs are outlined in the following sections.

3.12.1 Method initialization

The BMI library must perform several steps when it is first initialized. This is the point at which it must enable all of the active modules and initialize any internal bookkeeping information. The steps that it performs at this time can roughly be outlined as follows:

1. Parse the list of modules that will be used by the library.

2. Create a table to track each method and its instantiation of the method interface.

3. Load each method module one at a time and verify that it provides the correct symbols required by the method interface.

4. Create a reference list to use for mapping host addresses between various interfaces.

5. Initialize each method individually.

If any of these steps fails, then the initializer cleans up any work that it had done to that point and exits.

3.12.2 Posting and testing single operations

Posting and testing individual operations is relatively simple because it involves interacting with only one method at a time. These functions are carried out as follows:

1. Verify any arguments passed in by the user so that the method functions can assume that they are safe.

2. Search through the reference list for an entry that matches the address specified by the caller.

3. If such an entry is not found, return an error because the user specified address is invalid.

4. Find the method interface that matches the address (as indicated by the reference structure) and call the appropriate method to carry out the desired operation.

3.12.3 Aggregate operation tests

Testing operations from multiple methods within one library call is a a more complicated version of the scenario outlined in section 3.12.2. This situation must be handled by the BMI_testsome() and BMI_testglobal() function calls, which have the ability to interact with multiple operations at once.

In the BMI_testsome() case, the array of operations that the user submits is segregated based on which methods control which operations. Each method is then tested in a round robin manner to determine if any of those operations have completed. The results of the tests are aggregated back into a single array and returned to the user.

In the BMI_testglobal() function is similar except that the set of operations is not known in advance; therefore the initial sorting phase can be skipped.

The order in which round robin testing occurs is determined by the order in which the user previously specified the list of method modules at initialization time. Thus, high throughput interfaces can be given a somewhat higher priority by simply listing them first in the set of available method modules.

3.12.4 Address resolution

Address resolution is the final critical component of the method control layer. The following steps are performed in order to lookup an address based on a text description:

1. Search the reference list to determine if this lookup has already been performed. If so, immediately return the correct BMI address.
2. Contact each active method until one of them indicates that it was able to successfully parse the host name.
3. Create a new reference structure to track the address.
4. Fill in the reference structure with information about which module is responsible for it and which address handles may be used to refer to it.
5. Store the reference structure and return a valid BMI address to the user.

Note that since BMI is a connectionless interface, there is no way to guarantee that an address is truly valid until a user attempts to communicate with that host.

3.13 Bringing together the BMI library

The preceding sections outline all of the components that make up the protocol agnostic portion of the BMI architecture. All of these components are carefully constructed to limit each interface to only those components that need to access it. A user has no mechanism for directly interacting with method data structures or functionality, nor does any method have the ability to interfere with the operation of other methods. Modularity is preserved by only providing the absolute necessities of each interface to a given component. This helps to provide a safe environment for both the BMI user and the method implementer. It also insures that individual portions of the BMI architecture may be modified or optimized without interfering with core functionality. This is important in supporting future research.

The method control layer, support libraries, and user interface only provide a framework for the use of various network protocols, however. The real work of carrying out communications and emulating necessary network features is carried out within the BMI methods. The design of a set of prototype methods is outlined in the following chapters.

Chapter 4

BMI method case studies

For BMI to be successful, we must be able to efficiently implement methods for highly diverse networking environments. Thus we have chosen two distinct messaging systems as examples of the potential of the Buffered Message Interface. The first is TCP/IP. TCP/IP is a well known and widely adopted network protocol. It is a stream based and connection oriented, and provides full flow control and reliability. The second is GM. GM is a user level protocol that is supported on Myricom's Myrinet series of network adapters. GM is connectionless and provides no flow control.

These two protocols were chosen for two reasons. The first is their lack of similarity. If BMI can be shown to work efficiently over both protocols, then it should give at least some indication that the design is flexible. The implementation case studies will outline the challenges that were overcome for each method in more detail. Secondly, both protocols are mature and readily available on production level systems. Their behavior is well known and does not introduce any unexpected anomalies when studying network implementations.

Note that both implementations are essentially reference designs. More optimizations are available then have been implemented. Some of these have been identified for future work and will be noted where appropriate. The primary emphasis here is on obtaining dependable behavior and evaluating the Buffered Message Interface.

4.1 Method support libraries

Several support functions are provided for use by the BMI methods. These support functions were previously outlined beginning in section 3.11. The most important set of functions provide queuing capability. Almost every network method will utilize queues to keep track of network operations. Other support functions are provided to manage two important data structures: the method addresses and method operations. Method addresses are used as handles for hosts on the network. Operation structures are used to represent individual operations that are posted by the application. Both structures contain generic fields that apply to all methods while also allowing room for each method to store its own private information.

4.2 TCP/IP

4.2.1 Challenges

- **Connectionless emulation**: The TCP/IP socket interface is connection oriented. This means that a connection must be established between two hosts before communication can occur between them. It should also be terminated after all communication is complete. BMI, however, is a connectionless interface. The act of setting up and tearing down connections must be managed below the application interface, transparent to the application.

- **Data streams**: TCP/IP sockets operate as data streams. The interface has no explicit concept of message boundaries. Any amount of data can be sent or received in a single operation, but it may not necessarily complete at once.

Partial sends and receives are legal when using nonblocking TCP/IP sockets. This behavior does not match the BMI model of communications. BMI treats each message as an atomic unit that is either sent correctly or not.

- **Error recovery**: Recovery from communication failure using TCP/IP is somewhat complicated within the BMI environment. TCP/IP sockets contain a quite a bit of state because of the presence of a connection and a data stream. If a single operation fails, it is important to properly tear down any socket state and safely handle any further pending operations that intended to use the same socket.

4.2.2 Approach

Building blocks

The TCP/IP method implementation is based on three primary building blocks. The first is the sockio library, which provides a very small abstraction of the Unix sockets interface. It implements the basic operations, such as creating sockets, connecting sockets, setting TCP options, and performing read and write operations. The sockio library has been used extensively in the current PVFS implementation and is known to perform reliably.

The second building block is the socket collection library. The socket collection library provides a mechanism for managing groups of active sockets. Sockets are added to it as they are created, and removed from it as they are destroyed. The socket collection tracks all of the sockets and determines which ones can potentially send or receive data. When the TCP/IP method is prompted to do work, the socket collection is tested to determine which sockets are ready to handle operations. The internal polling set is dynamically resized as necessary to accommodate arbitrarily large collections of sockets. This interface also allows the possibility of implementing new polling strategies without compromising code that depends on the socket collections.

The final building block for the TCP/IP method is the operation queuing library provided by the method control layer. The queues are necessary to preserve the ordering of messages and to keep track of operations that are currently in progress. There are a fixed number of queues available that are used for communication with all hosts. This approach is chosen over maintaining separate queues for each host for reasons of simplicity and maintainability. We intend to show that the use of global method queues is not detrimental to performance as long as the queue management and searching is performed in an efficient manner.

Message modes

The TCP/IP method offers three modes of operation depending on the parameters of the message to be sent. The first, unexpected mode, is specified by the user when sending unexpected messages as outlined in section 3.3. The other two, eager and rendezvous mode, are transparent to the user and are internally chosen by the TCP/IP method based on the size of the data region to be transfered.

The semantics of each mode of operation are defined as follows:

- **rendezvous**: This is the simplest messaging mode supported by the TCP/IP method. It is chosen for larger messages (the default threshold specifies that rendezvous mode will be used for any message over 16 Kbyte in size). In this situation, the sender will first send a header describing the message and then immediately follow it with message data. The receiving method can accept the header at any time. However, it will not begin to read in the message data until the receiving user has posted a matching receive operation. This is why this mechanism is termed "rendezvous" mode; the bulk data transfer is not allowed to occur until both the sending and receiving user have prepared for the operation.

- **eager**: This messaging mode is chosen by the TCP/IP method for smaller message sizes (by default, less than 16 Kbyte in size). The sender (as in the rendezvous case) first transmits a header describing the message and then follows it with the actual message data. The receiver will accept this header and then make a decision about how the message data should be handled. If a matching receive operation has already been posted by the user, then the message data will be read into the receive buffer for that operation. If a matching receive has not yet been posted, then the receiving method will dynamically allocate a temporary buffer for the data to be stored in. This allows the operation to make progress even if the receiving user is not yet ready for the data. Once the matching receive operation is posted, the data can be copied into the final buffer and the temporary buffer is destroyed.

- **unexpected**: Unexpected messages are handled in an almost identical manner to eager messages. The only variation is that there is no final buffer copy step if the sender transmits data before the receiver is ready. Instead, the temporary buffer is passed to the user when the user checks to see if any unexpected messages have been received. The semantics of unexpected messages dictates the the receiver does not get a chance to specify the destination buffer for the message. It is created by the method.

It is interesting to note that in all three cases the sending method behaves the same. This seems counter-intuitive at first because it would appear that (at least in the rendezvous case) that the sender should wait until the receiver is prepared before transmitting the actual message data. However, we can rely on the natural behavior of the TCP/IP sockets interface in this context. TCP/IP sockets allow a sender to start sending data before a receiver is ready. The data is simply buffered at the operating system level until it can be transferred. This buffering of data improves overall latency and throughput because the operating system does not have to wait on user intervention to begin moving data once the opportunity arises.

Queuing model

As mentioned earlier, the TCP/IP method uses queues to keep track of network operations that are either in progress, awaiting resources, or completed. Six separate queues are used:

- **Send queue**: Contains all send operations (for any mode or address) that cannot be initiated yet. Operations are typically queued here because a previous operation to the same address has not yet completed. This queue ensures proper message ordering when multiple send operations are posted.

- **Completed queue**: Contains all operations that have completed transmission, whether successful or not. Operations are removed from this queue once the user has queried with the appropriate BMI_test function.

- **In-flight receive queue**: Contains a list of receive operations that have already begun but are not yet completed. It provides a fast mechanism for identifying where incoming data should be placed once a socket has data available.

- **Eager receive queue**: Contains receive operations that are small enough to be accepted in eager mode. These operations have not yet received any data. Once data begins to arrive for a particular operation, it will be moved to the in-flight receive queue.

- **Rendezvous receive queue**: Contains receive operations that will be accepted using rendezvous mode. These operations have not yet received any data. Once data begins to arrive for a particular operation, it will be moved to the in-flight receive queue.

- **Buffering receive queue**: Contains eager or unexpected receive operations which have begun buffering data before the user posted a matching buffer. This queue is searched when an eager message is posted to determine if the operation has already been completed.

Notice that there are more receive queues available than send queues. There are two reasons for this design. First of all, as noted earlier, all send operations are treated the same from the method's point of view. There is no distinction between TCP/IP sends that occur for different message modes. Secondly, the handling of receive operations requires more queue searching than in the send case. We therefore gain a measurable boost in performance by simply splitting up the queues to minimize search time.

Scenarios

The internal workings of the TCP/IP method can best be summarized with a few simple examples. The first example is an eager mode send operation, as shown in Figure 4.1. The operation begins with the user calling BMI_post_send(). The method will first check to see if there are any sends already scheduled for the same address. If it finds one, the message is immediately queued to preserve ordering. Otherwise, the method makes an attempt to send the message envelope and as much data as it can (without blocking) before queuing. If the message is completed at this time, it is never queued, and the return value of BMI_post_send() indicates its completion.

If the send does not complete on the first try, it will remain in the send queue until the internal socket collection indicates that work may continue on the operation. Since the TCP/IP method does not possess its own thread of control, it cannot do

Figure 4.1: TCP method (typical send scenario)

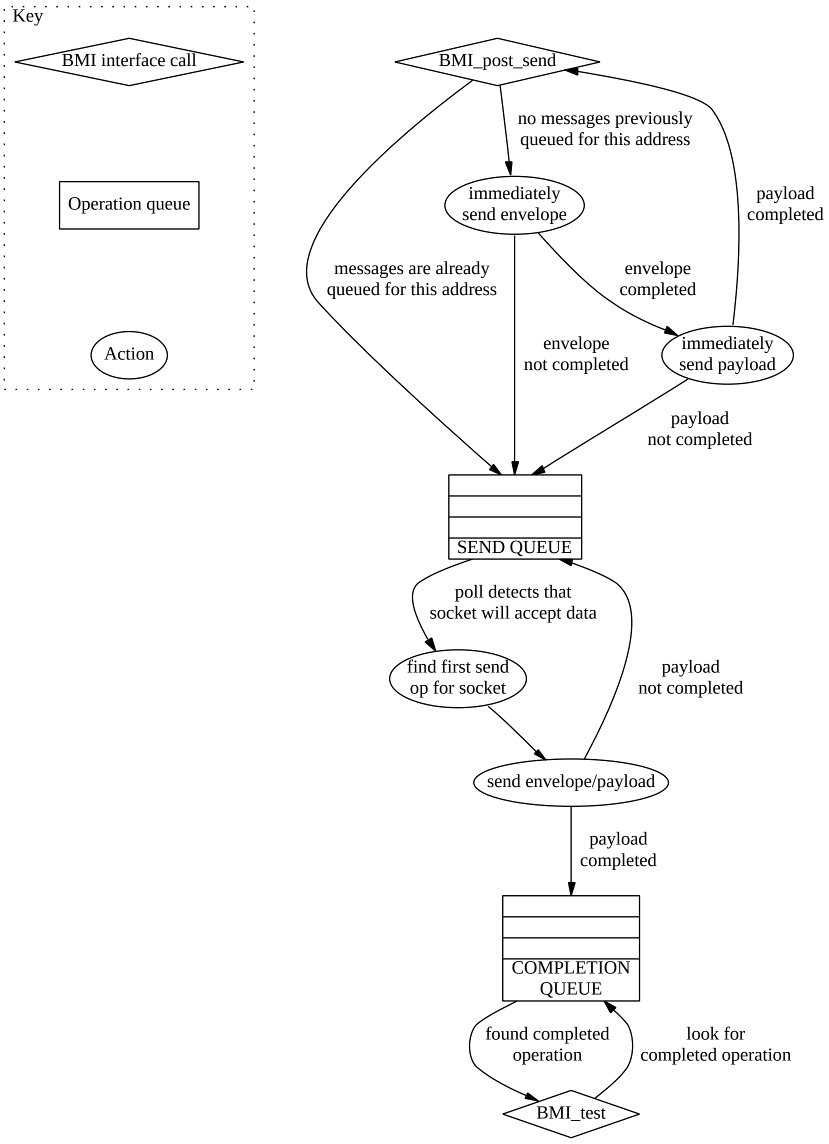
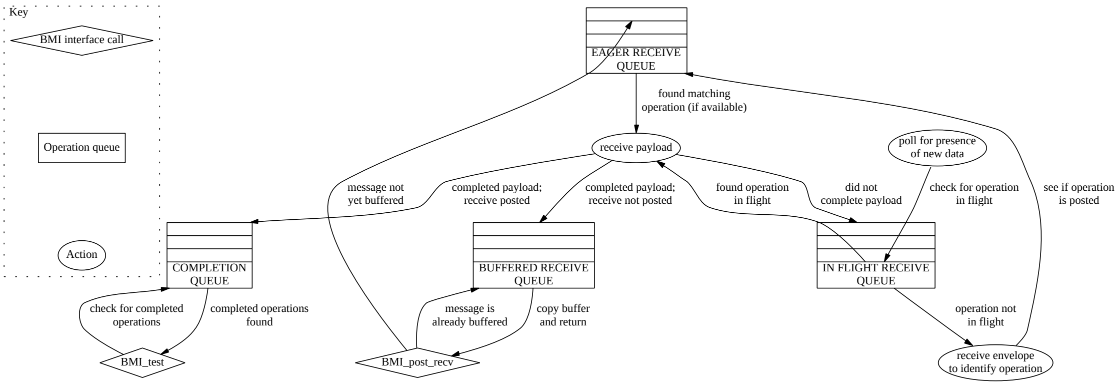


Figure 4.2: TCP method (typical receive scenario)



work until the user calls BMI_test(). Once an operation is finished, it is stored in the completion queue, where it will be recovered by the user in subsequent calls to BMI_test().

The common receive scenario is slightly more complicated than the send case. Figure 4.2 outlines the flow of events for an eager mode receive. When BMI_post_recv() is called by the user, it first checks to see if the receive has already been buffered. If so, it copies the payload out of the temporary buffer and returns immediately. Otherwise, the receive operation is queued.

When the socket collection indicates that data is available, the method first checks the in flight receive queue to see if the data belongs to an operation that is already in progress. If the method finds a match, it continues receiving data. If there is no matching operation in flight, then the method reads the envelope data from the socket to determine the parameters of the message.

If an operation completes before the matching receive has been posted by the user, it will be stored in the buffered receive queue. Otherwise, the operation is placed in the completion queue where it remains until the user tests to see if it has completed.

4.2.3 Possible optimizations

The approach outlined thus far for implementing a TCP/IP BMI method is sufficient to provide the correct messaging semantics. However, there are several optimizations which may be implemented to improve performance. Some of these techniques are presented below. A few of them will be explored more fully in section 5.

- **Nagle's algorithm**: Nagle's algorithm [20] attempts to improve efficiency on distributed networks by limiting the number of small packets in flight at any given time. If TCP/IP messages have been sent for which no acknowledgment has been received, then Nagle's algorithm prevents packets below a certain size threshold from being transmitted until all acknowledgments have been received. This may cause excessive delay for small messages if the network is reliable and does not need such a conservative approach to small messages. Therefore, most modern TCP/IP implementations have a mechanism for disabling this algorithm from the application level.

- **Eager receive opportunities**: Immediate completion of any network operation is beneficial to performance because it avoids the overhead of testing for completion later. The TCP/IP method does not yet take advantage of every such opportunity in the receive case. One example occurs if a receive is posted when no other receive is queued for that address. In this situation, it could take the optimistic approach and immediately check the matching socket for data. If the sender has already begun transmitting, then the receive operation can make progress before it is even queued.

- **Tuning socket buffers**: The amount of data that can be buffered during a send or receive is determined by the operating system's TCP/IP socket buffer size. This parameter may be specified globally for the entire system, or it may be specified on a per socket basis. If it is set on a per socket basis, then the method has the ability to adjust the buffer size dynamically in response to the type of operations being handled. It may be lowered to conserve system resources, or increased in order to allow the kernel to do more work per system call.

- **Reducing memory allocation**: Dynamic allocation of memory is typically an expensive operation. At this time, the TCP/IP method allocates several internal bookkeeping structures as needed, rather than reusing a pool of existing structures. This may be hindering small message latency in some situations. In particular, the method operation structure (which tracks the state of all network messages) is allocated for almost every message that is transmitted. This could be alleviated by simply replacing the method operation interface with an implementation that maintains a pool of structures that are already initialized and ready for use. Similar techniques are used in most operating system kernels to limit the overhead of acquiring common data structures.

- **Queue separation**: The range of available operation queues was chosen as a compromise between method complexity and queue search time. However, this may not be optimal for all work loads. The queues could be instrumented to provide more information about search times. This may indicate problem areas that could be targeted for improvement in queue layout.

- **Alternative polling algorithms**: The current implementation polls in the same order through the socket collection on each iteration. This strategy could be replaced with a round robin or other simple polling mechanism that more evenly distributes work among active sockets. We may also be able to identify situations in which more than one message can be placed in a single socket per iteration. Right now the test functions try to send or receive only one message per socket for each function call.

4.3 GM

4.3.1 Challenges

- **Memory management**: GM requires all data buffers to be pinned into physical memory before transmission. We can take advantage of this using the BMI memory management functions. However, we must also support the use of arbitrary user buffers at the BMI interface level. This requires either internal buffer copies or memory registration to maintain interface semantics.

- **Flow control**: GM does not provide any form of message level flow control. In fact, it requires that all receive buffers be posted before message transmission begins. Therefore, the BMI method must implement flow control in order to relax the message ordering semantics. BMI does not guarantee synchronization between sender and receiver.

- **Matching posted buffers**: There is no way to specify which incoming message will match a given receive buffer posted by the GM user. The user cannot specify that a buffer is only to be used for messages from a particular host, for example. Therefore, once a message is received, the method must analyze it to determine its origin and perform a memory copy if necessary to put the data in the correct user specified location.

- **Limited token resources**: The GM interface forces the user to refrain from posting sends or receives unless it possesses an appropriate token. There are a finite number of tokens. They are consumed when a send is initiated or a receive buffer is posted. The tokens are then returned when a send or receive completes.

4.3.2 Approach

The BMI GM method implementation relies heavily on the queuing mechanism provided by BMI to preserve message ordering and maintain the state of the system. It also takes advantage of several features specific to the GM library interface that assist in managing message completion.

Message modes

The GM method supports two messaging modes. The decision to use one or the other is based solely on the message size. The most complex messaging mode is the *eager handshake mode*. It is used by default for any message larger than 8 Kbyte, but this parameter is tunable. This mode is considered eager because it allows messages to be buffered at the receiving side before the user posts a matching receive operation. However, the sending and receiving hosts must negotiate at a low level before transmitting the message.

Smaller messages are sent using *immediate mode*. Immediate mode requires no handshaking at all. The sender assumes that the receiver is always prepared to receive messages of this size.

Methods are capable of sending and receiving control messages independent of actual message data. These control messages may contain flow control information, handshaking information, or actual message data in the case of immediate mode messages. When the GM method is initialized, its first task is to provide a large number of receive buffers for accepting control messages. These buffers are reused after processing control messages and are replaced as quickly as possible to ensure that there are always buffers available to handle new control messages.

When a sender initiates an eager handshake communication, it first sends a control message to the receiver to announce that it wishes to transmit. The receiver then prepares a buffer of the appropriate size for receiving the message. Once the buffer is ready, it sends a control message back to the receiver in response to announce that the buffer is ready. The actual message payload is then transfered.

When a sender initiates an immediate communication, it simply sends the data as payload on a control message. The receiver will accept these messages without negotiating in advance.

Flow control

The GM method implements very basic flow control mechanisms. The actual communications mechanism on Myrinet networks are extremely reliable, so the goal of flow control is not to avoid congestion or lost packets, but rather to conserve memory resources. Only a finite number of receive buffers may be posted at a given time, and one must be careful not to exhaust the memory resources of a host machine.

There are two types of buffers which may be posted by a receiving host. The first is the large data payload buffer used during eager handshake communications. The use of these buffers can be easily controlled by the receiver since they are only used in this specific mode. Once the receiver processes a control message requesting a buffer of this type, the receiver has the option of waiting as long as it needs to before posting the buffer. If it runs out of memory resources, it simply delays sending a control response until the resources are available. This prevents a sender from transmitting the payload too quickly.

Control message buffers are the second resource that must be conserved. The GM method attempts to keep as many of these available as possible, but it is still possible for a client to overrun the available buffers by sending small messages faster than they can be processed. In order to prevent this situation, a limit is placed on the number of send messages that may be in flight between hosts at any given time. A message is considered to no longer be in flight once the sender is sure that the receiver has processed it. The number of messages allowed per host is tunable. Increasing this parameter improves performance because it allows deeper pipelining if many messages are allowed to be transmitted back to back. However, this parameter must be reduced in larger networks to ensure that each host can accept messages from every other host simultaneously without exhausting memory resources.

Message pipelining

Note that *pipelining* is used extensively in eager handshake mode. There are several steps to carrying out eager handshake messages. One message may be in step one while another message is in step two, and so on. The available resources determine how many messages may carry out the same step at the same time. If one resource is exhausted, other steps are allowed to continue up until stalling on that resource. This approach is very similar to instruction pipelining in modern microprocessors [22].

Retransmission

Even with the above flow control scheme, the scalability of the GM method is limited. With enough hosts on the network, any finite number of available buffers could be consumed in a degenerate case, such as a many to one communication. Therefore, the method must be able to recover from packet loss that occurs when messages are sent before buffers are ready.

The GM library provides extensions to detect and recover from this situation. However, the documentation for these extensions is incomplete at the time of this writing because these features have undergone modifications during recent release cycles. Implementation of a retransmission policy for the BMI GM method is therefore being postponed for future work.

Scenarios

The GM method can be understood more fully by observing a few scenarios. The immediate mode messages are just a simplified version of the eager handshake messages, so we will focus on eager handshake mode. Keep in mind that this method is not implemented using threads. Therefore, the state machine is only driven when the caller invokes BMI function calls.

A state diagram of the send case is shown in Figure 4.3. When the application posts a send buffer, the method first checks to see if the data needs to be copied into a suitable buffer. Once the buffer is ready, it must check three resources before continuing. First it makes sure that a send token is available. Then it checks to see if any messages are queued ahead of it. Finally, it checks to see how many messages are already in flight to the target host. If any requirement is not met, then the message is queued. Otherwise, it continues by sending a control request to the target host.

If the target host responds and grants permission to send the data payload, then the method must again either obtain a send token or queue the message until a token is available. Finally the data payload is sent and the operation is placed in a completion queue to be recovered when the application calls BMI_test().

The matching receive example is shown in Figure 4.4. When a control request is received from a sending host, the method must obtain a token to receive the data payload. If no token is available, the operation is temporarily queued. A buffer is then posted to accept the payload, and a control response is sent to inform the sender that it may proceed. The operation is then queued until the data arrives.

Once the data arrives, the method checks to see if a matching receive operation has been posted in the control match queue. If so, the operation is finished and moved to the completion queue to be recovered during a BMI_test() call. If not, the operation is placed in the receive post queue to wait until BMI_post_recv() is called.

Figure 4.3: GM method (typical send scenario)

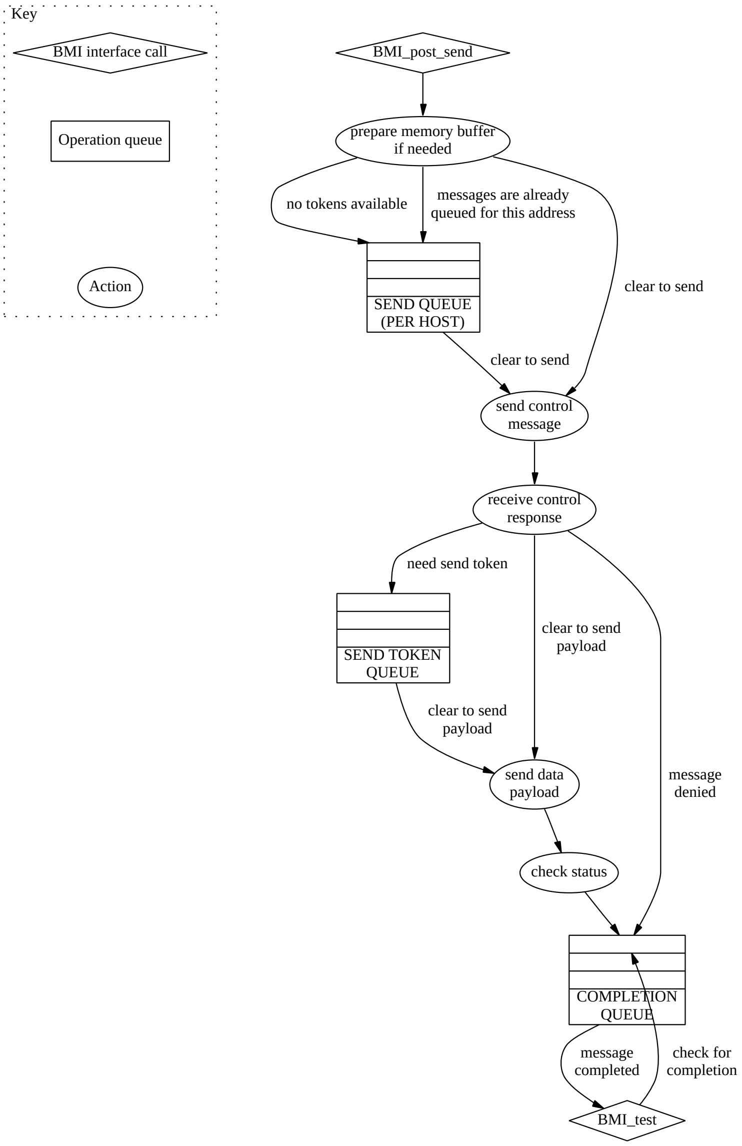
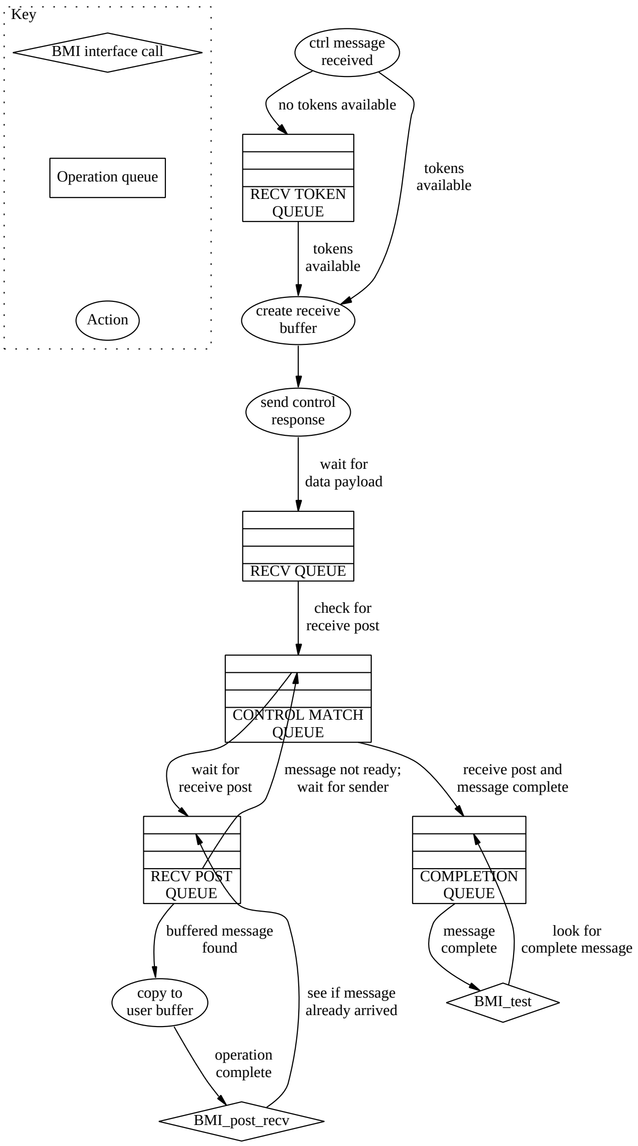


Figure 4.4: GM method (typical receive scenario)



BMI_post_recv() will complete immediately in this case because the data will already be available before the function is invoked.

4.3.3 Possible optimizations

The GM messaging library provides tools for implementing many communication optimizations. The original GM method implementation is focused on correctness rather than speed, so several of these possibilities have been left for later work:

- **Registering memory**: All memory buffers transmitted using GM must either be allocated by the GM library or registered by the GM library. Currently, if a BMI user passes in a buffer that was not preallocated, a new buffer is created and the data is copied into it. This is done so that extra room may be provided in the buffer for storing header information. However, it would probably be much faster to register the existing buffer. Allocating room for the header could be avoided by transmitting the data using control messages.

- **Fast GM messages**: GM optimizes for very small messages by transmitting them within the low level GM control packet. If the application elects to handle this type of message, it can operate directly on the data buffer from the GM control packet. If it ignores this type of message, then the data is transparently copied into a normal GM buffer and handled just like any other message. This buffer copy could be avoided if the method processed this special type of message (called "fast" messages by GM).

- **Avoiding memory copy on receive**: Since the GM method does not allow the caller to specify which incoming messages are placed in which buffer, incoming messages are always copied to ensure that they end up in the correct user buffer. In some cases, this is unnecessary because the caller doesn't actually care where the data is placed. We could take advantage of this by allowing a special type of receive operation that provides the buffer to the caller once the operation completes, rather than specifying it in advance. This would require a modification to the BMI interface.

- **Advanced flow control**: The BMI GM method implements a very rudimentary flow control scheme. A more advanced approach could potentially yield a speedup, but is beyond the scope of this document.

- **Pooling control buffers**: As mentioned earlier, control receive buffers are pooled and reused once they are processed to avoid the overhead of allocating new memory regions. This is not done in the send case, however, because it is more difficult to locate available buffers when they are needed. A simple lookaside list implementation could resolve this issue, however, and avoid extra memory allocation time.

- **Reducing memory allocation for bookkeeping**: The BMI methods allocate several structures to track pending operations and network addresses. If these structures were allocated in advanced and reused as needed it would help to cut down on message latency.

Chapter 5

Results

We must measure the performance of the BMI implementation in several scenarios to evaluate its efficiency. MPICH will be used for comparison purposes so that there is a point of reference for observing the results. MPICH was chosen because it is capable of providing almost all of the messaging ability that BMI possesses, and there is an MPICH device available for both TCP/IP and GM (the two BMI reference implementations).

In all cases, the benchmarks were implemented using MPI functions that most closely match the capabilities of BMI. However, since MPI is a much broader implementation, there are often MPI functions available that would be more optimal. These functions are ignored for this comparison, however, because we are more interested in general purpose baseline performance.

It must be emphasized that this is not a direct comparison of the two interfaces. In general they solve very different problems and thus are subject to different design constraints. BMI has an advantage in these tests because it is a much less featureful interface and the test applications do not always take advantage of the most optimal MPI application approach.

The first analysis of BMI performance will focus on point to point bandwidth, round trip latency, and many to one and one to many communications. The first two classes of tests will hopefully point out fundamental strengths and weaknesses of the interfaces, while the latter tests will attempt to evaluate BMI in situation more similar to what would occur in a real life file server implementation.

5.1 Test environment

These tests were all carried out on the Chiba City scalable cluster at Argonne National Laboratory [7]. The cluster was configured as follows at the time of our experiments. There were 256 nodes, each with two 500-MHz Pentium III processors, 512 Mbytes of RAM, a 100 Mbits/sec Intel EtherExpress Pro Fast Ethernet network card operating in full-duplex mode, and a 64-bit Myrinet card (Revision 3). The nodes were running Linux 2.4.2. There were two MPI implementations: MPICH 1.2.1 for Fast Ethernet and MPICH-GM 1.2.0 for Myrinet. None of the tests were performed using more than 65 nodes at a time.

5.2 Initial TCP/IP results

All TCP/IP tests were performed using the Ethernet network on Chiba City with MPICH 1.2.1. In addition, baseline bandwidth measurements were performed using the ttcp test utility, version 1.12 [25]. The ttcp utility operates directly on TCP/IP sockets with no abstraction layer. It should therefore give a good indication of the maximum obtainable TCP/IP bandwidth.

5.2.1 Bandwidth

Bandwidth was measured by transmitting a predetermined amount of data between two hosts using a variety of message sizes. The data was marked with a pattern before timing began so that its correctness could be verified after receipt. All messages were posted in order before testing for completion. Timing for both sender and receiver hosts includes the time required to post and test for completion of all messages. MPI tests were performed using the MPI_Isend(), MPI_Irecv() and MPI_Test() functions. BMI tests were performed using BMI_post_send(), BMI_post_recv() and BMI_test().

Memory buffers were not allocated using the BMI interface in any TCP tests. This sort of allocation has no impact on performance in the TCP/IP method because TCP/IP has no mechanism for optimizing message buffers.

All figures shown are the result of averaging five measurements. The BMI and MPI tests were run within seconds of each other in each case in order to ensure that the system was in a consistent state for each test.

Figure 5.1 shows the TCP/IP bandwidth as measured from the sending host for very small message sizes, ranging from 100 bytes to 1000 bytes. The total amount of data transfered in every case was 1,000,000 bytes, or nearly one Mbyte. This means that the total number of message sent for each data point ranged from 10,000 to 1,000 messages.

The raw TCP performance (as measured by ttcp) shows negligible impact from choice of message size. However, both MPI and BMI demonstrate the penalty imposed from an extra layer of abstraction over the sockets interface. BMI does not reach its peak bandwidth capability until the message size is 500 bytes or greater. These results also show that the BMI interface imposes an overhead of about 5% compared to direct socket communications for message sizes larger than 500 bytes. MPI does not reach its peak capacity within this message range.

Figure 5.1: Small message TCP/IP bandwidth (send)

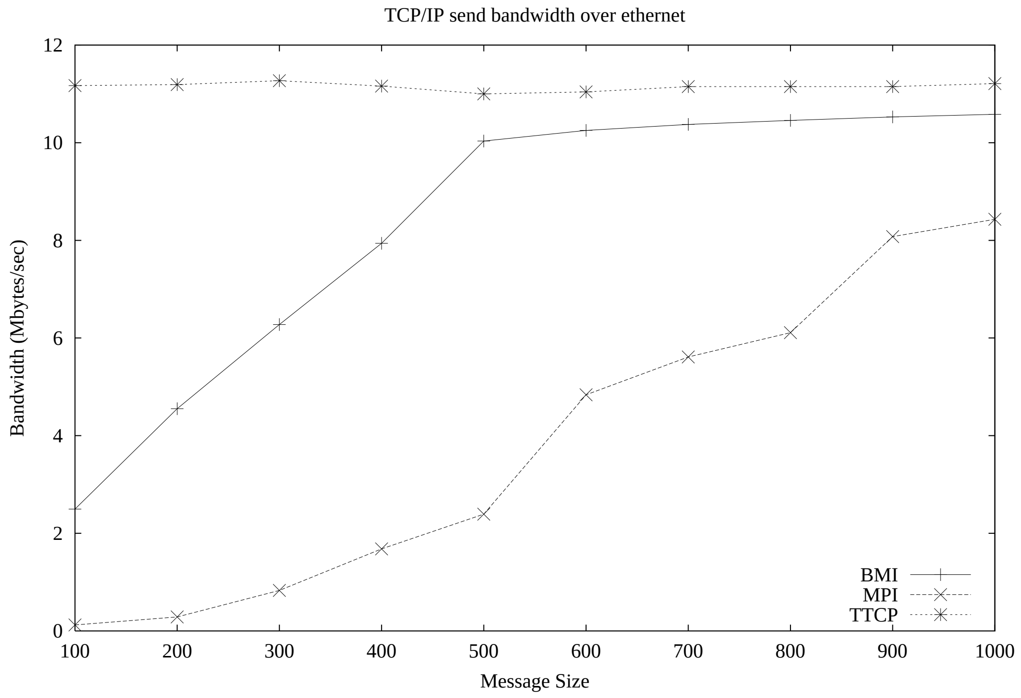


Figure 5.2: Small message TCP/IP bandwidth (receive)

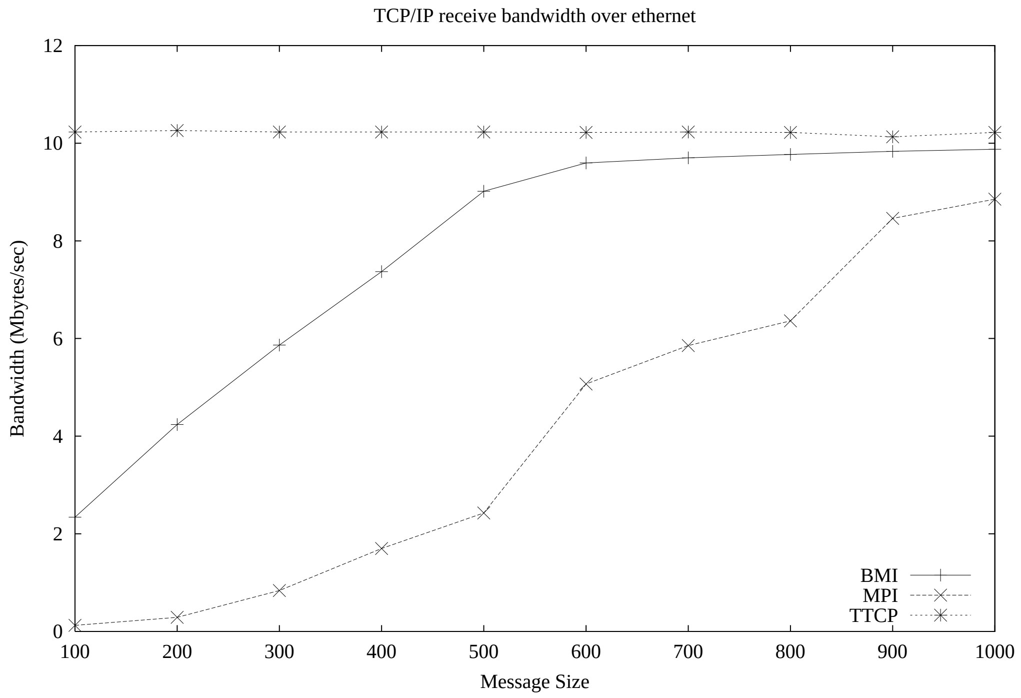


Figure 5.3: Larger message TCP/IP bandwidth (send)

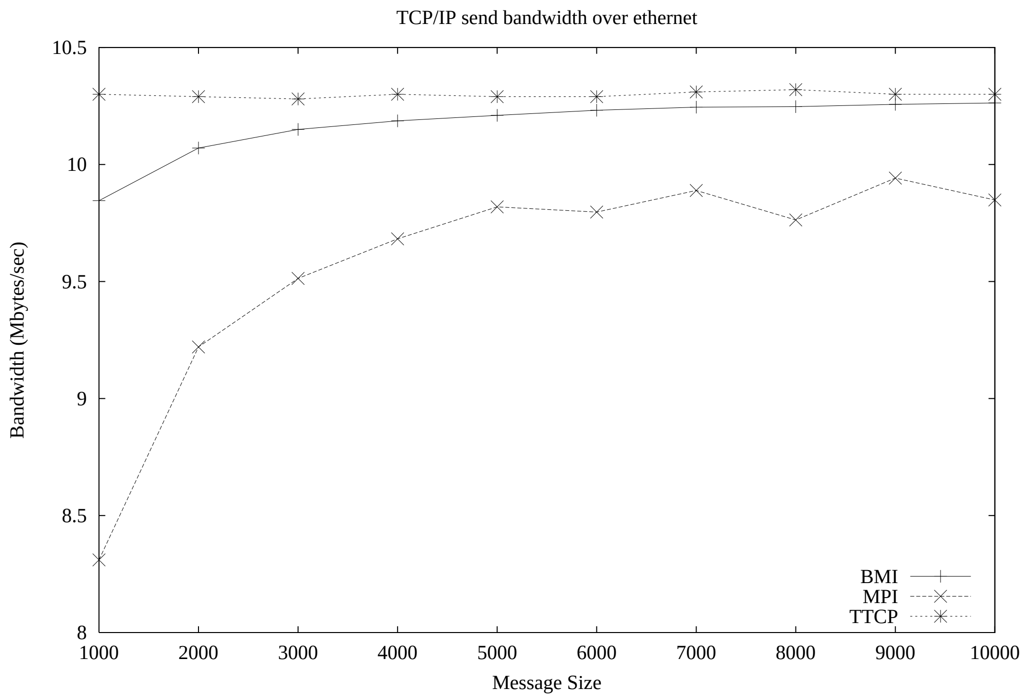
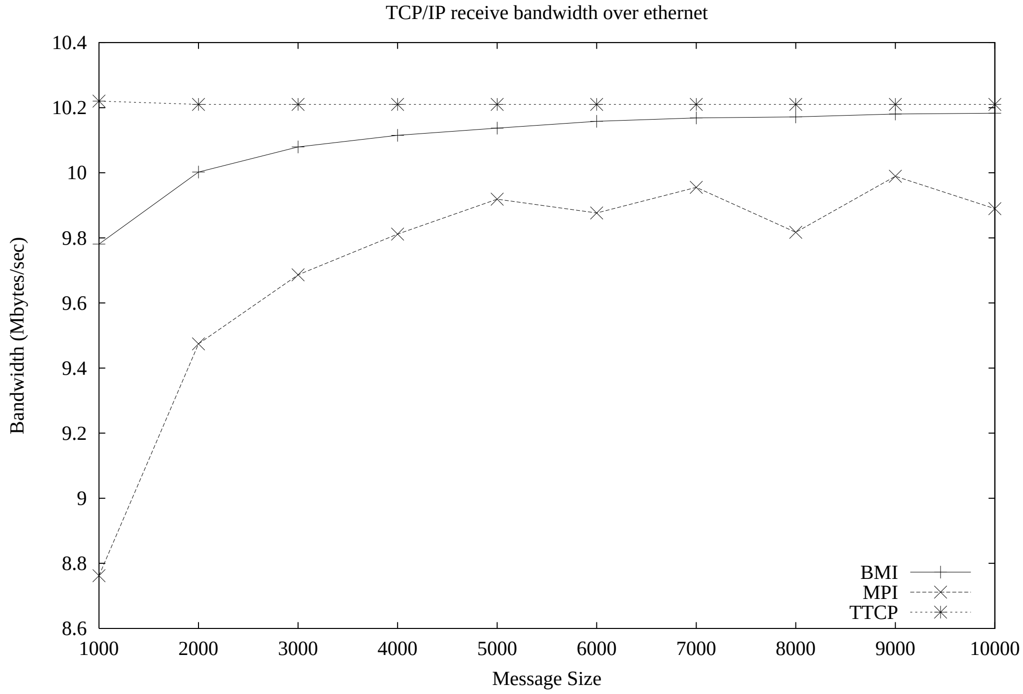


Figure 5.2 shows very similar results for bandwidth measured from the receiving host.

Figures 5.3 and 5.4 show bandwidth results for the transfer of 10,000,000 bytes (nearly 10 Mbytes) of data, using message sizes ranging from 1,000 bytes to 10,000 bytes. This means that the number of message needed to complete the transfer varied between 10,000 and 1000, just as in the previous case.

Notice that with 10,000 byte messages, the BMI method only obtains about 9.8 Mbytes/sec for a 10 Mbyte transfer, as opposed to 10.75 Mbytes/sec for the 1 Mbyte transfer shown in the previous graphs. This shows that both message size and number of messages sent have an impact on performance. Again, the interface is imposing a noticeable overhead when a large number of messages are queued up.

Figure 5.4: Larger message TCP/IP bandwidth (receive)



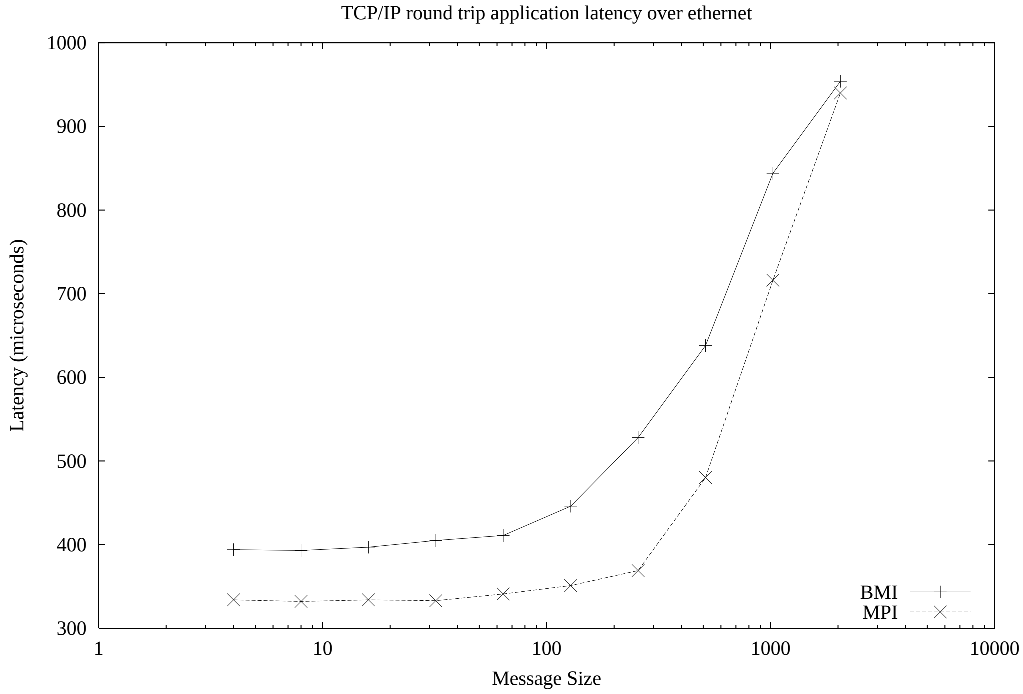
As the message size gets larger and the total number of messages being sent decreases, the BMI performance begins to match the raw TCP/IP performance. For large messages the overhead imposed by the interface is less than 1%.

5.2.2 Latency

Latency results were obtained by measuring the round trip transmission time of a message between two machines. All communications were done using nonblocking function calls. Timing started just before posting the first send operation and ended just after the receipt of the response message completed successfully. All measurements were carried out from the application level.

Figure 5.5 shows the round trip latency as measured using both BMI and MPI. The message sizes ranged from 4 bytes to 4 Kbytes (on a logarithmic scale). The MPI latency is much lower than the BMI latency in this test, though they begin to

Figure 5.5: TCP/IP round trip latency



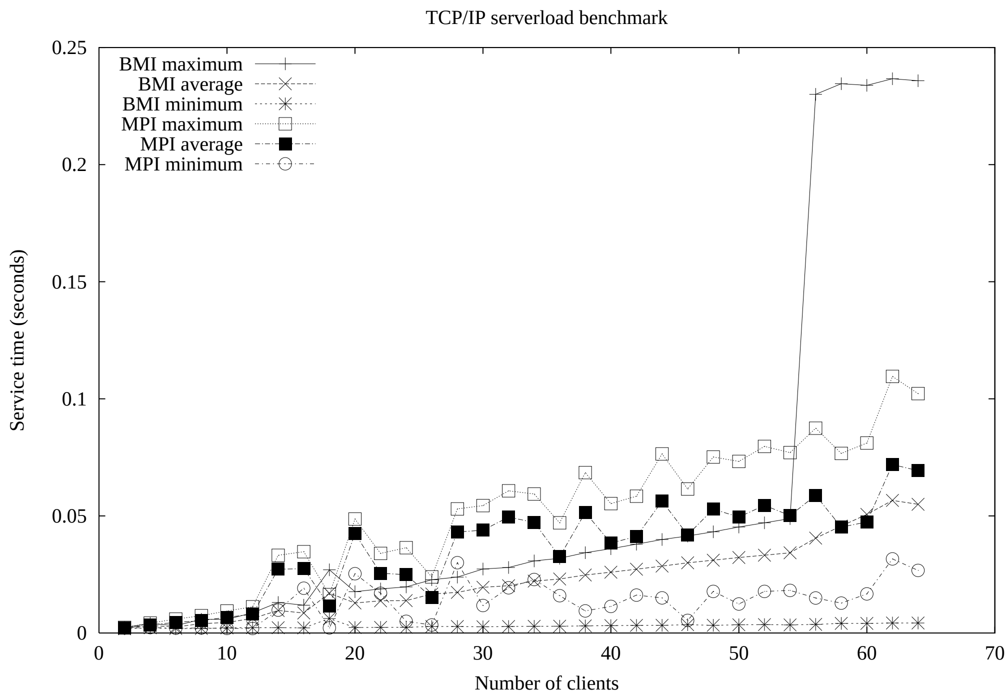
converge for larger message sizes. Section 5.4.1 explores techniques for lowering the latency found in the BMI TCP/IP method.

5.2.3 Simulated server load

This test is structured as follows. A single host is setup to listen for MPI or BMI messages. When it receives a message (formatted as a small request), it replies by sending another message of the requested size back to the sender. The clients that communicate with it are synchronized using MPI so that they all attempt to contact the server simultaneously. This is intended to measure BMI performance under a load that resembles what would happen if many clients were to contact a single server simultaneously for a small data read operation.

In addition, the server is given no advance knowledge of which clients will contact it, nor in what order they will communicate. The BMI portion of the server deals

Figure 5.6: TCP/IP many to one performance (10K messages)



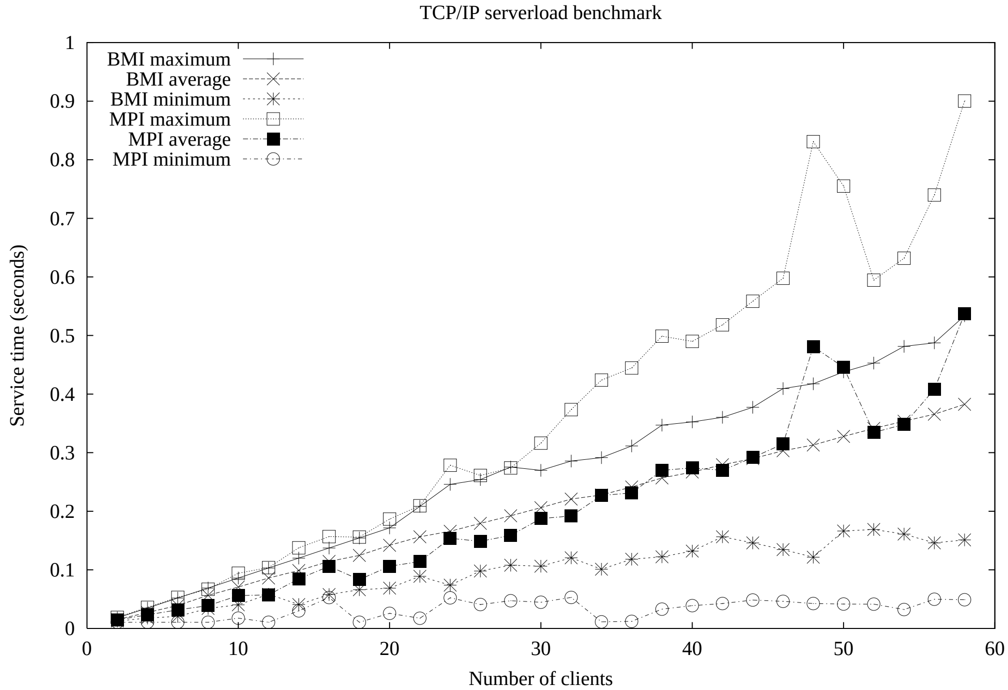
with this by using the BMI unexpected message facility. The MPI portion of the server handles it by posted a collection of receives that will match a wildcard (any) sender.

Timing information (round trip application latency) is measured from the client side. The average, maximum, and minimum times among all clients are averaged across five test runs. Figure 5.6 shows the results of this test using 10 Kbyte messages for anywhere from 2 to 64 clients.

The BMI method performs well for this message size and exhibits a very stable performance curve.

Figure 5.7 shows results from the same test scenario, but with a message size of 100 Kbytes rather than 10 Kbytes. The interesting behavior from this test case is that the BMI and MPI tests exhibit very similar average performance across the range of clients. However, the MPI interface obtains this average through a much wider range

Figure 5.7: TCP/IP many to one performance (100K messages)



of individual client measurements. MPI obtains much lower latency for some clients while also obtaining much higher latency on other clients when compared with BMI.

5.3 Initial GM results

All GM tests were performed using the Myrinet network on Chiba City with MPICH-GM 1.2.0. All tests were setup in an identical manner to those presented in section 5.2, but using the GM method for BMI and Myricom's MPICH-GM implementation for MPI.

5.3.1 Bandwidth

The bandwidth was measured again by transmitting a fixed amount of data using a variety of message sizes. The BMI GM method has the ability to take advantage of optimized buffers, so the performance was also measured with the buffers being allocated in advance using BMI_memalloc(). In test cases with this BMI preallocation, the memory allocation time was not included in the timing. This is consistent with the normal test cases in which the time needed to malloc() the data buffers was not included in communication timing. All data points shown are the result of averaging five test runs.

The original intent was to measure performance over the same range of message sizes used in the TCP/IP tests. However, it was discovered that the MPICH-GM implementation was incapable of completing the 1 Mbyte bandwidth test using 100 Byte message sizes. The test generally failed with memory allocation errors.

As a result, performance measurements had to be taken from a larger message size range to provide data points from both interfaces.

Figure 5.8 shows bandwidth as measured for the transfer of a 1,000,000 byte region using message sizes ranging from 1,000 bytes to 10,000 bytes. This resulted in a total number of messages ranging from 1,000 to 10,000.

Both MPICH-GM and BMI were configured by default to switch into three way handshake mode at the 8 Kbyte message size. This can clearly be seen in Figure 5.8 from the jumps in the curve. BMI performance (both for normal and preallocated memory) dropped off at 8 Kbytes, while the MPICH-GM performance improved at 8 Kbytes. The MPI performance below 8 Kbytes was surprisingly poor.

Figure 5.9 results from the same test run as measured from the receiving side. The results are very similar, except that MPICH-GM performance doesn't improve quite as much on the 8 Kbyte boundary as it did in the send case.

Figures 5.10 and 5.11 show the bandwidth as measured for the transfer of 10,000,000 bytes of data using messages sizes ranging from 10,000 bytes to 100,000 bytes. The BMI method plateaus at about 27 Mbytes/sec if the memory buffers are not allocated in advance. The steps carried out for this transfer are identical to those done

Figure 5.8: Small message GM bandwidth (send)

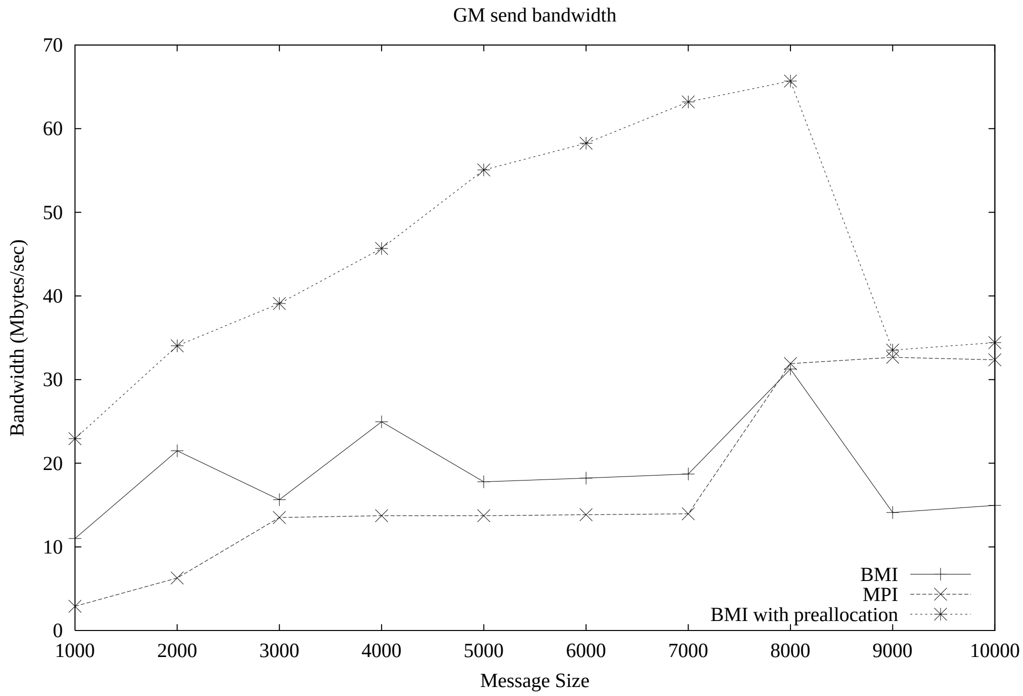


Figure 5.9: Small message GM bandwidth (receive)

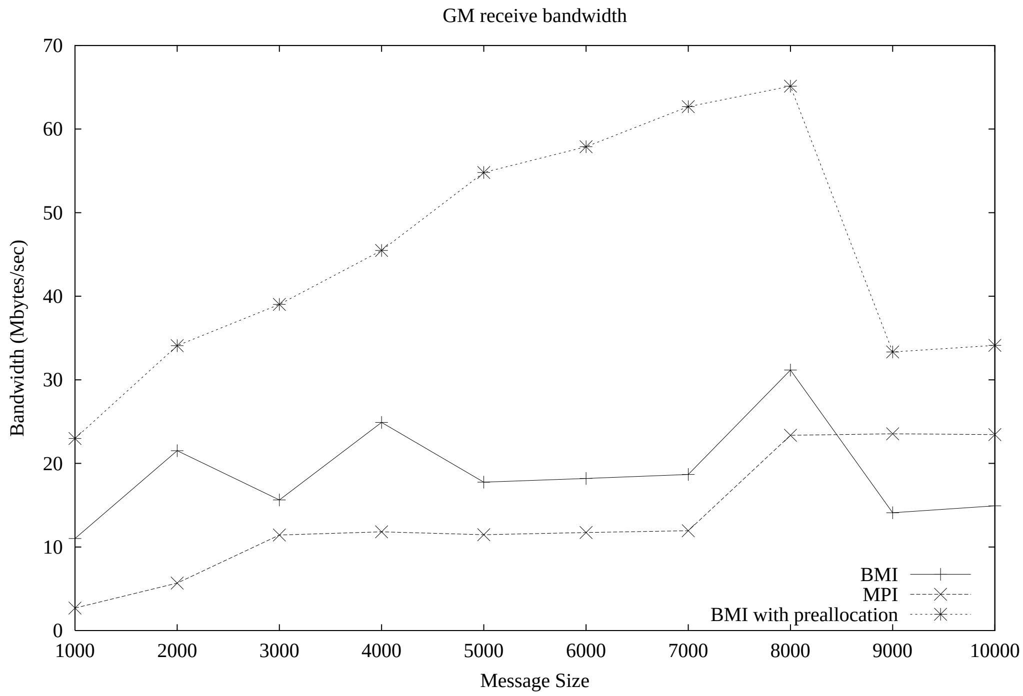


Figure 5.10: Large message GM bandwidth (send)

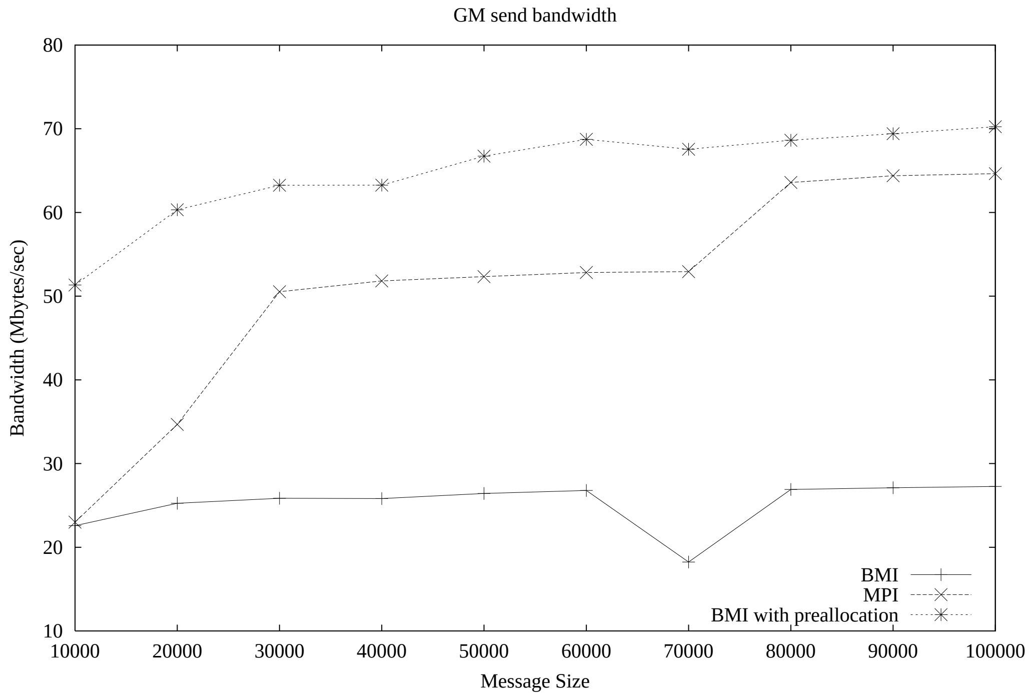


Figure 5.11: Large message GM bandwidth (receive)

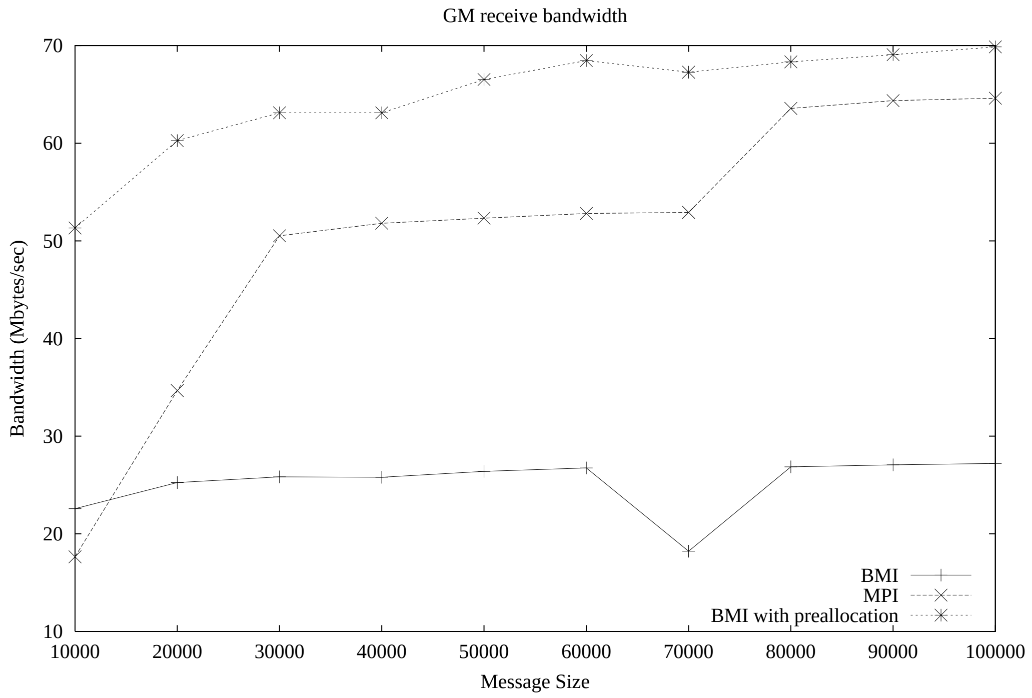
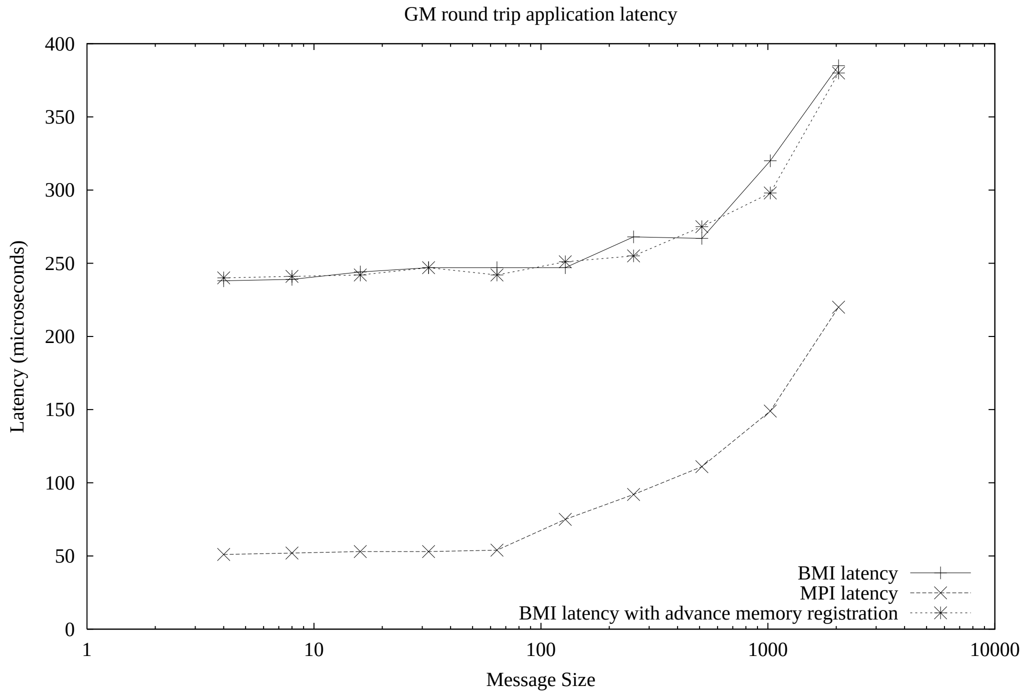


Figure 5.12: GM round trip latency



with preallocation except that an extra memory copy is incurred for every message. The penalty introduced by this approach is quite evident here. Section 5.4.2 explores methods of reducing this overhead for memory buffers that are not allocated in advance.

5.3.2 Latency

Round trip application was measured for GM in the same manner as done in the preceeding TCP/IP tests. The results are shown on a logarithmic scale in Figure 5.12.

Preallocation of memory buffers had negligible impact on overall performance, which demonstrates that the additional memory copy is not terribly expensive for buffers in this size range.

Figure 5.13: GM many to one performance (10K messages)

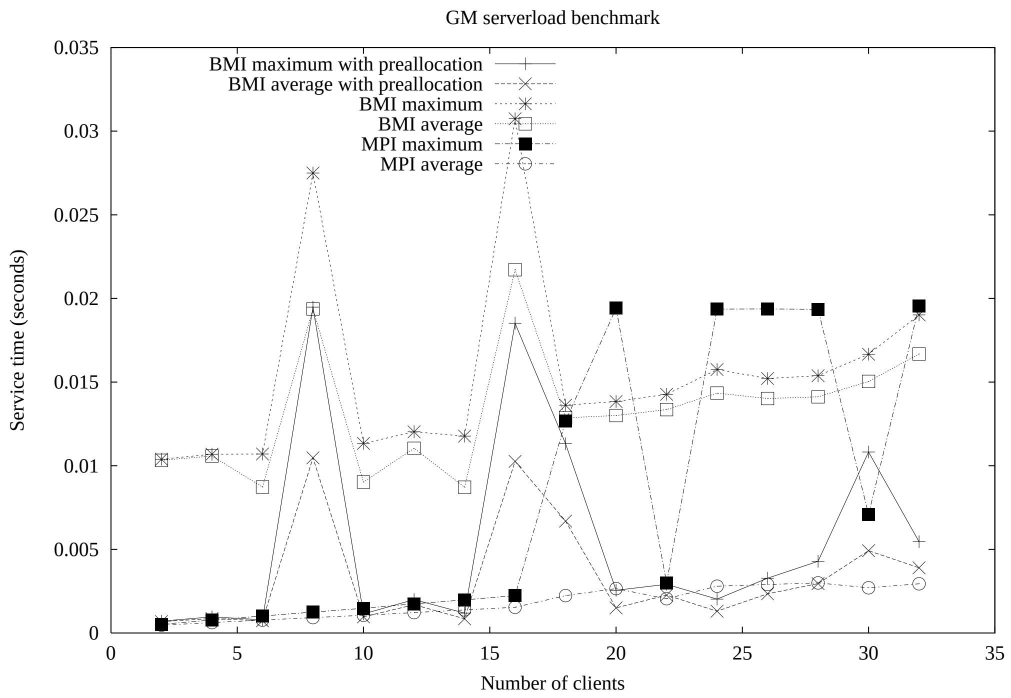


Figure 5.12 also shows that the BMI method was unable to obtain the low latency figures demonstrated by MPI at any message size. The discrepancy is quite large, with the BMI approach taking at least four times as long as the MPICH-GM implementation.

5.3.3 Simulating server load

The serverload benchmark was executed over GM following the same parameters as in the TCP/IP tests in section 5.2.3. Figure 5.13 shows the results of this benchmark for 10,000 byte message sizes.

Performance was erratic for all three cases (MPI, BMI, and BMI with buffer preallocation). The MPI performance was very similar to the BMI performance with preallocation. The BMI performance without preallocation was found to be substantially slower.

Figure 5.14: GM many to one performance (100K messages)

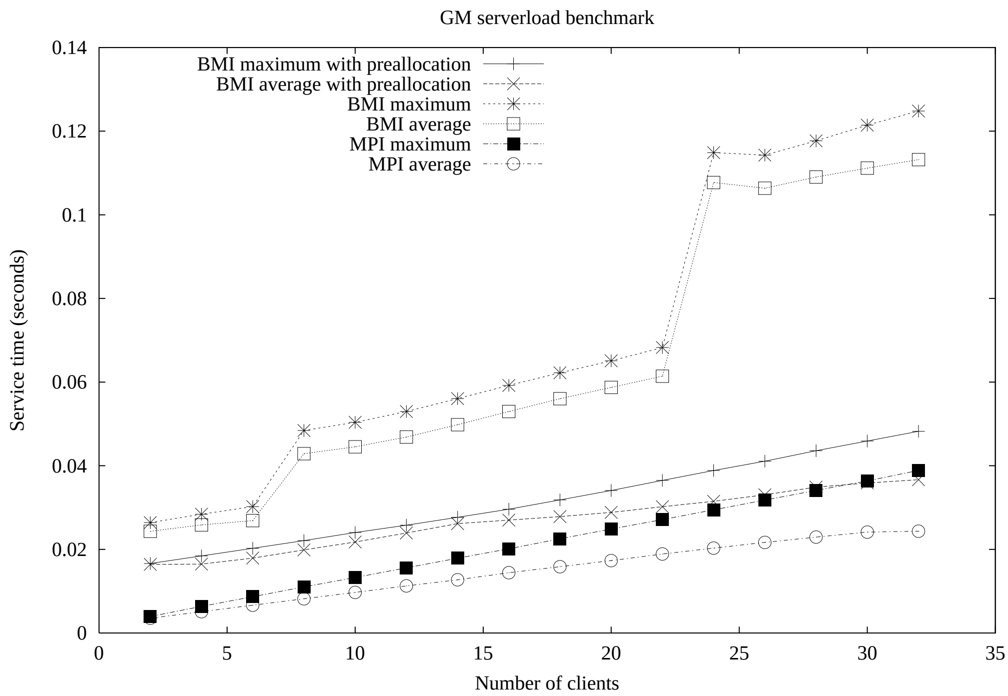
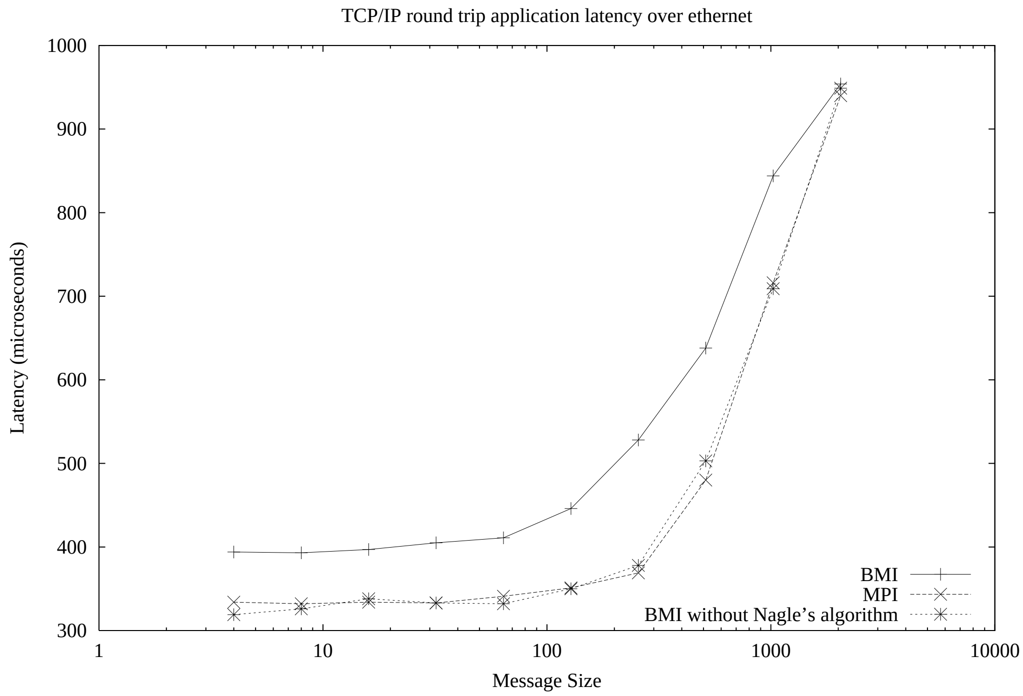


Figure 5.14 shows the results of the same tests when using 100,000 byte message sizes. The performance is much more predictable in this case. Again, the MPI performance exceeded that of BMI. The margin was especially large when the BMI buffers were not preallocated.

5.4 Evaluating problem areas

The initial BMI performance results indicate that some aspects of BMI are not reaching the potential offered by the underlying communications systems. In this section we will attempt to analyze and address these issues.

Figure 5.15: TCP/IP round trip latency



5.4.1 TCP/IP method latency

Figure 5.5 indicates that the BMI method is exhibiting relatively poor performance in terms of latency. The MPICH TCP/IP implementation is as much as 80 microseconds faster in round trip application measurements.

In order to find the source of this problem, the MPICH implementation was analyzed first. It was discovered that MPICH disables Nagle's algorithm by default on BSD based systems that support this option. This option is controlled by the TCP_NODELAY flag in the setsockopt() function. See section 4.2.3 for more information about Nagle's algorithm.

Figure 5.15 shows the results of the same latency test with the TCP_NODELAY option set for all sockets controlled by the BMI method. MPI and BMI exhibit almost identical behavior with this approach.

Figure 5.16: small message TCP/IP bandwidth (send)

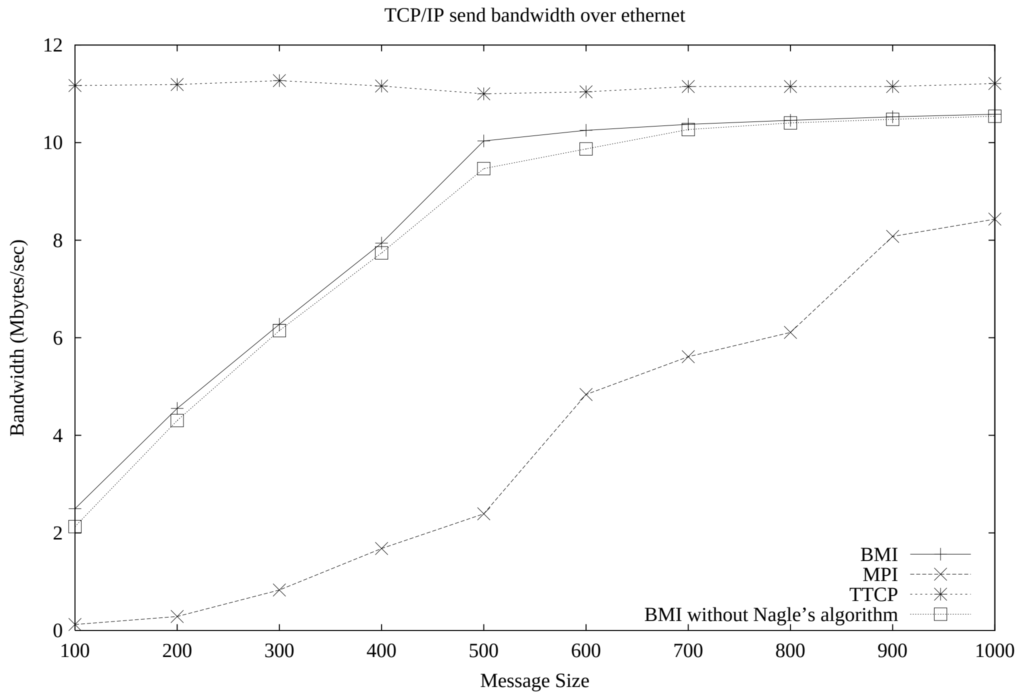
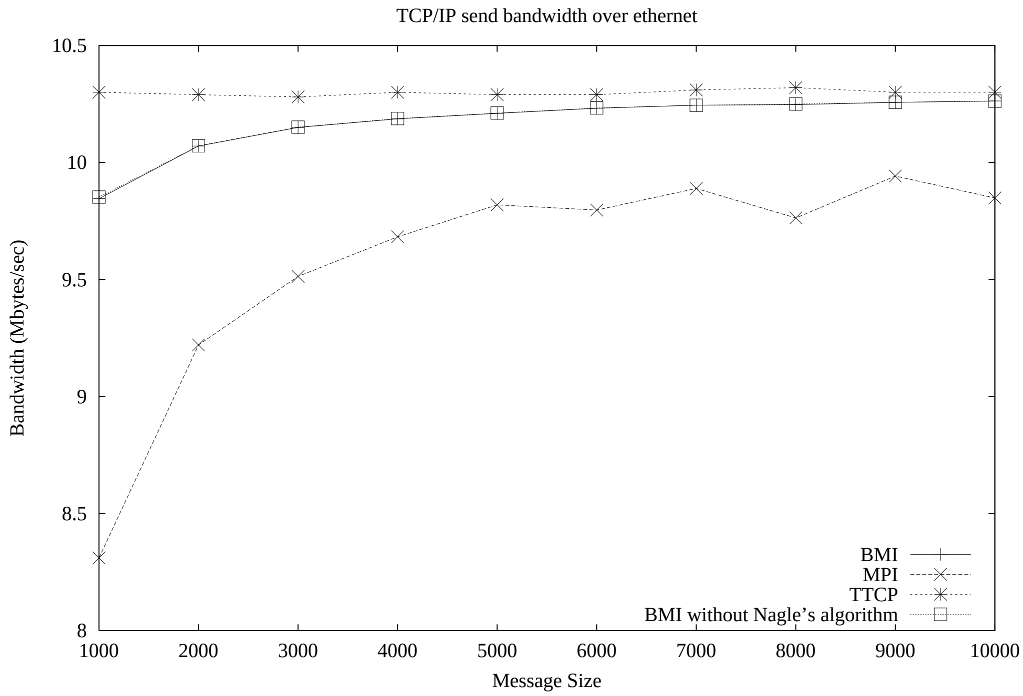


Figure 5.17: larger message TCP/IP bandwidth (send)



The raw bandwidth tests were also reevaluated to see if this approach impacted BMI bandwidth results. Figures 5.16 and 5.17 show that this option has minimal effect on the overall TCP/IP bandwidth. The only measurable impact occurred for messages smaller than 700 bytes. In order to reduce even this impact, it may be possible to implement an adaptive delay policy. Such a policy could elect to disable or enable Nagle's algorithm on a per socket basis depending on the characteristics of the messages that are queued up to be sent for that socket.

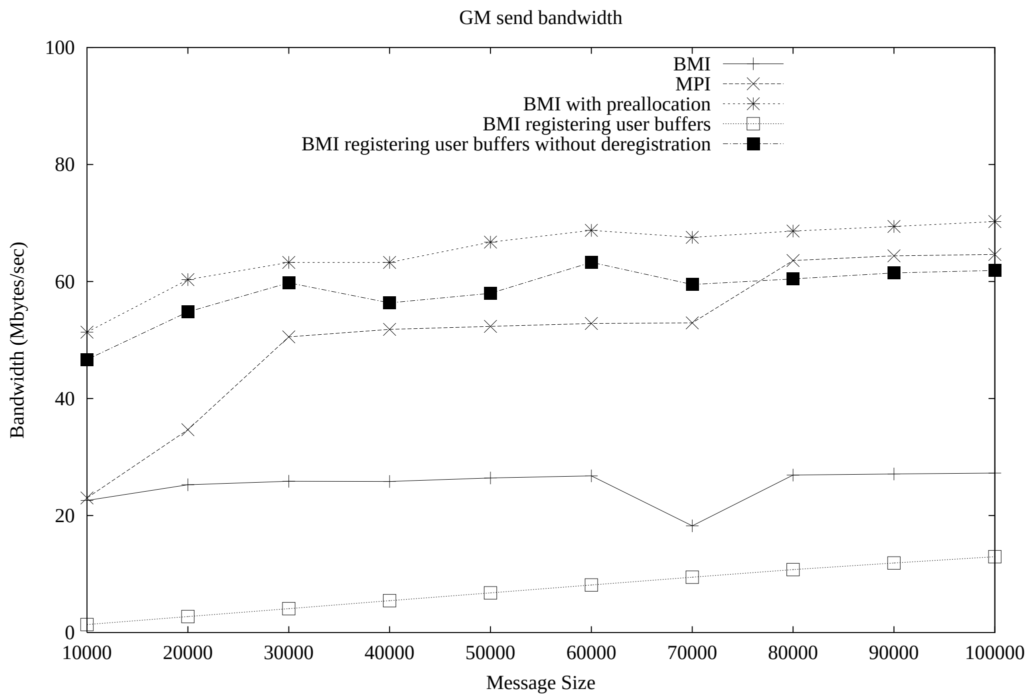
5.4.2 GM bandwidth for large messages

Initial bandwidth measurements for the GM method indicate that performance was quite good *if user buffers were allocated in advance using the BMI library.* However, if existing user buffers were utilized, then the performance was limited to around 27 Mbytes/sec (see Figure 5.10). This is not acceptable. The MPI performance figures indicate that a much higher performance can be obtained without using preallocation.

As explained in section 4.3.1, the GM library requires that all buffers to be transmitted be located in regions of memory that have been prepared in advance for DMA purposes. The initial BMI implementation met this requirement by allocating new DMA-able buffers for each message and copying the user buffer into it. This copy operation clearly was consuming a relatively large amount of time for large message sizes.

The alternative is to actually register the existing user buffers using the gm_register_memory function call. Buffers registered in this manner must later be released using the gm_deregister_memory function. One drawback to this approach is that low level BMI header information can no longer be piggy-backed onto the message data because there is no way to append contiguous memory to the message. However, this header information is not necessary for BMI messages that are transmitted using a

Figure 5.18: Large message GM bandwidth (send)



handshaking protocol, because the control messages contain a complete description for the message.

To explore this possibility, the BMI GM implementation was modified to register user buffers that had not been allocated using BMI. Each buffer was registered when posted, and then deregistered on completion. This would avoid the extra memory copy step. However, Figure 5.18 shows that the performance for this approach was absolutely terrible (see the "BMI bandwidth registering user buffers" data points).

The MPICH-GM implementation was then inspected to discover Myricom's solution to this problem. As it turns out, the MPICH device uses a special memory management system that registers user buffers as needed, but does not necessarily release them upon message completion. The buffers remained registered when returned to the user. This optimization is intended to be helpful if buffers are reused quickly.

The MPICH device keeps up with which memory regions have been registered, and only deregisters regions if system memory resources run low.

The BMI bandwidth benchmark used for the preceding experiments only uses 10 Mbyte of system memory per host. Therefore, for experimentation purposes it is possible to disable deregistration entirely to observe the impact. Figure 5.18 ("BMI bandwidth registering without unregistering user buffers" data points) shows the result. The performance was *much* higher, and in fact exceeded MPI bandwidth for most message sizes shown. Note that each buffer was used only once, so reuse of buffers was not an issue.

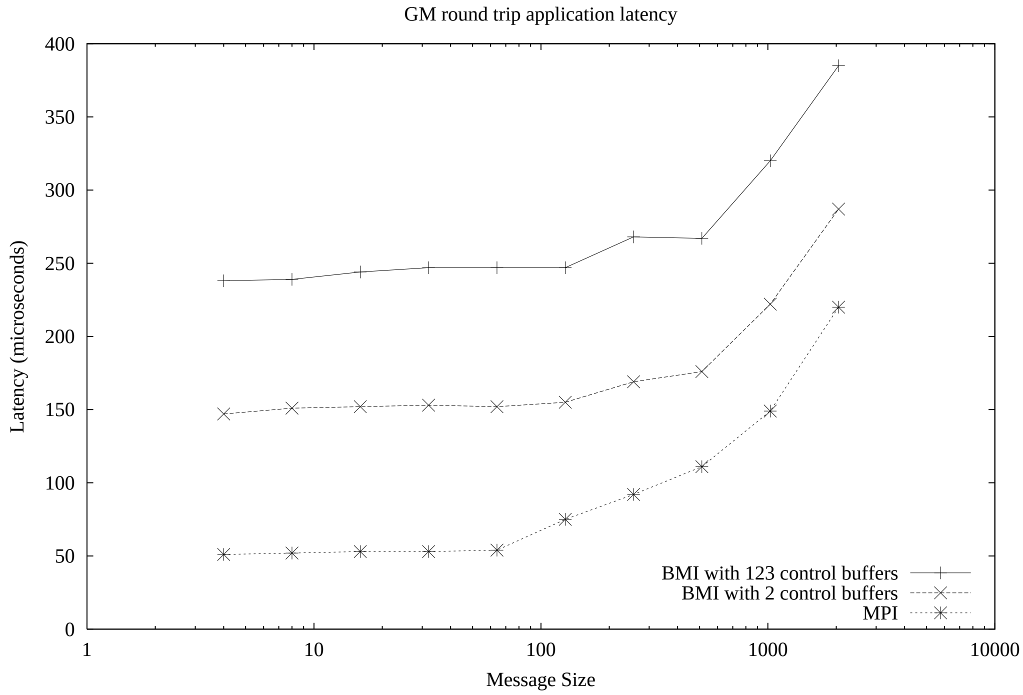
This experiment revealed that the act of deregistering GM memory buffers is extremely high overhead, which was an unexpected result. In order for BMI bandwidth with existing buffers to achieve the same level of MPICH, the BMI method must implement a similar memory management library which takes a lazy approach to deregistering system memory.

Note that receive buffers in the BMI GM implementation are always copied. There is no easy way to relax this constraint, due the messaging system's inability to specify the role of each receive buffer posted. Each time a message is received, it must be analyzed to determine which user buffer it matches. The data is then copied to that buffer.

5.4.3 GM method latency

Earlier GM method measurements (particularly for the round trip latency and server load applications) show that the latency of the implementation is not reaching its expected potential. The MPICH-GM implementation is as much as four to five times faster in terms of latency.

Figure 5.19: GM round trip latency



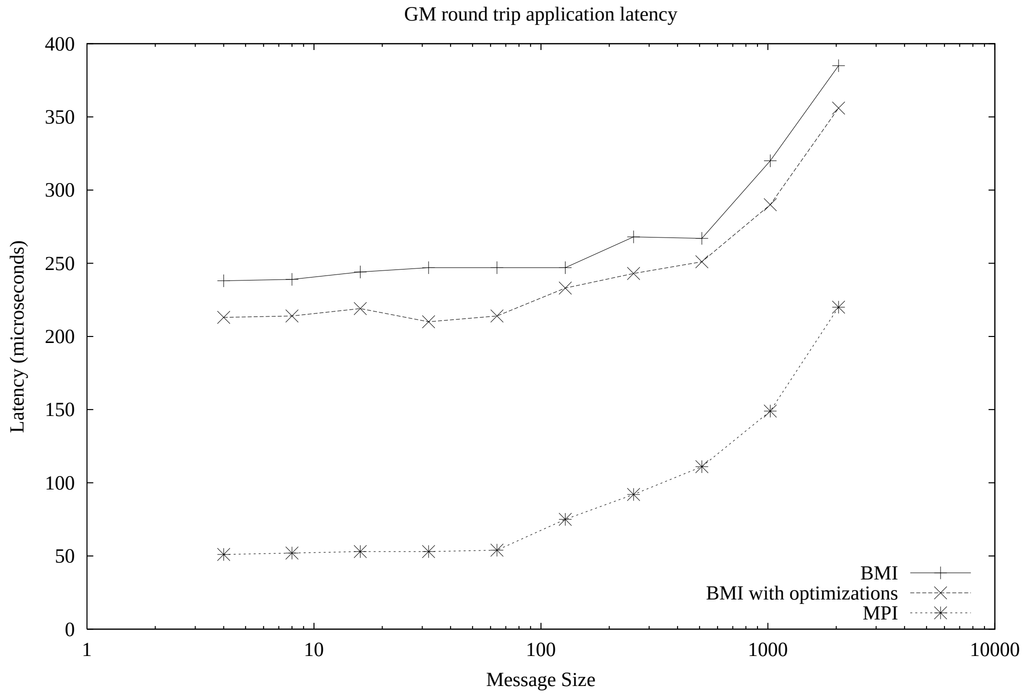
We therefore set out to investigate this discrepancy. A variety of optimizations were attempted, ranging from alternative GM interface functions to optimized allocation of internal structures. A few interesting results were found.

Note that the GM method implementation (like the MPICH-GM implementation) requires that a collection of receive buffers be posted as soon as the method initializes. This ensures that it will be able to accept control messages from its peers. It was discovered that tuning the number of buffers that were provided had a significant impact on latency, as shown in Figure 5.19.

The extra line in this graph indicates performance if only 2 buffers were posted in advance, rather than the typical 123. Note that the time required to actually post these buffers is not included in these measurements.

The difference in performance is nearly 100 microseconds. This margin is about *double* the total round trip time of the MPICH-GM implementation. This is odd

Figure 5.20: GM round trip latency



because the MPICH-GM implementation defaults to posting 123 buffers as well, and clearly it does not suffer from the same performance penalty that the BMI implementation does. This seems to indicate that there is a subtle difference in how MPICH-GM and BMI methods approach communication management that is still to be discovered. This may be caused by an operation that scales poorly with the number of posted buffers that should be avoided.

Note that a production quality method implementation *must* provide more than two control buffers at startup time. Otherwise the user risks the danger of overrunning the amount of available buffers when too many control messages flood the network.

The algorithm that determines how much work should be done during a BMI_test() function call was also found to have a significant impact on latency. The original approach was to only perform one gm_receive operation per cycle. The implementation was modified so that after each call to gm_receive(), gm_receive_pending() was also called to determine if there was any more work that could be immediately performed. If so, gm_receive() was called repeatedly (up to a bounding limit) until the event queue was emptied. This cuts down on the overall number of function calls to the BMI interface per communication. The results of this optimization are shown in Figure 5.20. This trimmed around 20 microseconds from the total round trip time.

Despite these discoveries, the BMI method still displays inferior results in the latency sensitive applications. The servload application is sensitive to this shortcoming and thus will not be re-evaluated until a solution is found to the latency problem.

Chapter 6

Conclusion

We presented the Buffered Message Interface in order to meet the need for a network abstraction layer for implementing parallel file systems on Linux clusters. This interface provides a simple application interface for accessing all the features of a communications network necessary for high performance I/O.

BMI demonstrates that a modular mechanism can be built and used effectively for communicating over various dissimilar networks. None of the performance testing applications were recompiled to adapt to the protocols used. All that was necessary was the presence of the proper BMI module and (in this case) a configuration file indicating which module to use.

The BMI interface was also capable of communicating with multiple dissimilar protocols simultaneously. The performance testing of this feature is beyond the scope of this document, but the semantics are implemented correctly.

We also observed that the efficiency of this implementation was on par with expectations in the majority of the tested scenarios. We hope that the obvious deficiencies will be addressed in future work on the method modules.

BMI will be a key component of the forthcoming Parallel Virtual File System version 2. PVFS2 is being designed with collaboration between Clemson University, Argonne National Laboratory, and Goddard Space Flight Center to provide the next step in parallel I/O technology to Linux clusters. BMI will insure that this implementation keeps pace with trends in networking technology without the need for redesign of the core file system. Once this new PVFS implementation has arrived, we will be able to evaluate the performance and usability of BMI within the context of an actual file system implementation.

6.1 Future work

There is still much research and development to be performed within the Buffered Message Interface. This work revolves around three critical areas: improvement of existing protocol methods, expansion into new protocols, and bringing the current implementation up to production level availability.

6.1.1 Improvement of existing methods

Both the TCP/IP and GM method implementations were quite successful. However, both messaging systems have room for improvement. Sections 4.2.3 and 4.3.3 list several possible optimizations, only a few of which have been realized thus far. The latency performance of the GM module is of particular interest after the results shown in section 5.4.3.

6.1.2 Expansion of supported methods

TCP/IP and GM methods were implemented as a proof of concept for dissimilar networks. There are several other method implementations that could be explored, however. Some possibilities include shared memory, VIA, and UDP methods. Each of these should be feasible within the previously defined Buffered Message Interface. VIA in particular should be relatively straightforward to implement because it shares several overall concepts with the GM interface. A shared memory method would perhaps be most interesting in terms of proving BMI's success in abstraction because it does not fall into the broad category of message passing communication. UDP would interesting because it would prove the feasibility of implementing reliable delivery within a BMI method.

6.1.3 Scheduling

BMI was designed so that it will be possible to couple it with a higher level component capable of making scheduling decisions. This is of particular interest in I/O server design. A scheduling mechanism should be able to obtain load information from BMI by using the get_info() function. Policy hints may be provided using the set_info() function. This approach remains untested at this time, however.

6.1.4 Production level availability

The Buffered Message Interface is relevant not only as a research project but also as a production level component of a true parallel file system. This places added emphasis on its ability to provide production level robustness. In particular, it must be very resilient (or at least very predictable) in the face of individual network errors. A file system cannot tolerate deadlock or critical failure of the underlying network subsystem.

Extensive stress testing will be necessary to bring BMI to production level quality. All of the tests performed in this document were done with an emphasis on measuring performance. More rigorous testing should examine degenerate cases and forced network failures to ensure that the interface is robust.

Bibliography

[1] The Dolphin SCI interconnect white paper. Technical report, Dolphin Interconnect Solutions, Inc., February 1996.

[2] Donald J. Becker, Thomas Sterling, John E. Dorband Daniel Savarese, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, 1995.

[3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet a gigabit per second local area network. *IEEE Micro*, 15, February 1995.

[4] Philip Buonadonna, Andrew Geweke, and David Culler. An implementation and analysis of the Virtual Interface Architecture. In *Proceedings of SC98*, Orlando, Florida, November 1998.

[5] Bill O. Callmeister. *POSIX.4: Programming for the Real World*. O'Reilly and Associates, Inc., Sebastopol, CA, 1995.

[6] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[7] Chiba City, the Argonne scalable cluster. `http://www.mcs.anl.gov/chiba/`.

[8] G. Chiola and G. Ciaccio. GAMMA: a low-cost network of workstations based on active messages. In *Proceedings of PDP'97 (5th EUROMICRO workshop on Parallel and Distributed Processing)*, London, UK, January 1997.

[9] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[10] Emulex IP storage networking. `http://wwwip.emulex.com/ip/index.html`.

[11] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[13] M. Halem, F. Shaffer, N. Palm, E. Salmon, S. Raghavan, and L. Kempster. Can we avoid a data survivability crisis? *Science Information Systems Newsletter*, (51), 1999.

[14] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[15] W. B. Ligon III, S. P. McMillan, and R. B. Ross. Tuning TCP performance in beowulf computers. Technical report, Parallel Architecture Research Laboratory, Clemson University, 1999.

[16] M-VIA: A high performance modular VIA for linux. `http://www.nersc.gov/research/FTG/via/`.

[17] Message Passing Interface Forum. MPI documents. `http://www.mpi-forum.org/docs/docs.html`.

[18] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[19] Myrinet software and documentation. `http://www.myri.com/scs`.

[20] John Nagle. Congestion control in ip/tcp internetworks (rfc-896). Technical report, IETF Network Working Group, January 1984.

[21] Avneesh Pant, Sudha Krishnamurthy, Rob Pennington, Mike Showerman, and Qian Liu. VMI: An efficient messaging library for heterogeneous cluster communication. Technical report, National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign, 2000.

[22] David A. Patterson and John L. Hennessy. *Computer Architecture; A Quantitative Approach (2nd Edition)*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[23] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings, IEEE Aerospace*, 1997.

[24] Robert B. Ross. *Reactive Scheduling for Parallel I/O Systems*. PhD thesis, Electrical and Computer Engineering Dept., Clemson University, 2000.

[25] Test TCP. `ftp://ftp.arl.mil/pub/ttcp/`.

[26] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In

Bob Hertzberger, Peter Sloot, editor, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer-Verlag, April 1997.

[27] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[28] VI architecture. `http://www.viarch.org`.

[29] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Int'l Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.